Lecture Summary

In this lecture, we looked at two implementations of the List data type from last lecture: using linked lists and arrays. In particular, we paid attention to the running time of the key operations. We then briefly looked ahead at Stacks and Queues which we will see in depth next lecture.

In the previous lecture, we talked about the List data type. Basically, the idea was to have an “array” that allows us to access elements by index, but can also dynamically grow or shrink when we insert or remove elements from it. Specifically, the functions we wanted (written here as a pure abstract class, to specify an abstract data type) were the following:

```cpp
template <class T>
class List {
public:
    T get (int pos) const = 0;
    // Returns the element at position pos, assuming it is within bounds of the list.

    void set (int pos, const T &newvalue) = 0;
    // Overwrites position pos with newvalue, assuming pos is within bounds of the list.
    // Does not expand the array if pos is out of bounds;

    void remove (int pos) = 0;
    // Removes the element at position pos, assuming pos is within bounds of the list.
    // Shifts everything to its right one position to the left.
    // In the process reduces the size of the list by 1.

    void insert(int pos, const T &value) = 0;
    // Inserts the element value at (right before) position pos,
    // assuming pos is within bounds (or one larger than the size).
    // Shifts all elements from pos onward one position to the right.
    // In the process expands the list size by 1.
};
```

So the only functions that change the size of the list are remove and insert, and they change it by 1. The textbook version also has a few additional functions, but these are the ones we care about here. (All the rest is just icing.)

There are two natural ways to implement this abstract data type: as a linked list, or as a dynamically allocated array which grows as needed. We will explore both of the implementations, and their advantages and disadvantages.

1 Implementation using a linked list

In implementing a List as a linked list, the one functionality we need crucially is to find the element corresponding to position \( i \). Then, we can read, overwrite, delete or insert there quite easily.
The idea for finding the element is quite simple: we start from the head, follow the pointers, and count how many steps we have taken so far. Then, we stop after the right number of steps. So we add the following (private or protected) function to our inheriting class LinkedListList (or whatever we want to call it):

```cpp
IntListElement *locate (int i) {
    IntListElement *p = head;
    for (int j=0; j<i; j++) p = p->next;
    // at the end, p points to element i
    return p;
}
```

Notice that this function will most likely segfault (or cause some other kind of error) if the list does not actually contain $i$ element. Ideally, whatever function is looking for a position should always check to make sure the position is in bounds before calling `locate`. The easiest way to do this would be for the implementation to contain a private variable `int length` which contains the current number of elements in the list. That variable can easily be kept updated. An alternative would be to change the termination condition of the loop to `(j<i && p != NULL)`. This will avoid the segfault, and return a NULL pointer when the index is out of bounds. The problem is that we may then not be able to diagnose where the problem is coming from.

Now that we have the `locate` function, implementing the other functions becomes pretty straightforward:

- **get (int pos)**: We simply look for the correct element, then return its data field, as follows:

  ```cpp
  T LinkedListList::get (int pos) {
    if (inRange(pos)) return (locate(pos))->data;
  }
  ```

  This assumes that we have already written a (private or protected) function `inRange`. It’s not clear what we should do if `pos` is not in the correct range. C++ forces us to return something. The correct mechanism — which we will learn about in a few lectures — is to use `exceptions`.

- **set (int pos, const T & newvalue)**: We first use `locate` to find the right pointer (assuming that `pos` is in range), then simply overwrite its `data` field with `newvalue`.

- **remove (int pos)**: We first use `locate` to find the right element, and then use the function `removeElement` which we implemented as part of our `LinkedList` implementation. Also, we decrease our stored value for the size of the list.

- **insert (int pos, const T & value)**: Here, we first use `locate` to find the pointer `p` to position `pos` (assuming that it is in range, of course). Then, we create a new `IntListElement q` whose `data` field contains `value`. The `prev` and `next` pointers should then be set correctly as follows:

  ```cpp
  q->prev = p->prev; q->next = p;
  p->prev->next = q;
  p->prev = q;
  ```

  Of course, this needs to be adjusted a bit if the insertion is supposed to happen before the first element (`pos==0`) or after the last element (`pos==size`). In those cases, we also need to update the `head` and/or `tail` of the linked list.

---

1We could instead expand each node of the list to contain its position in addition to a `next` pointer and the data. But that would not be a good idea: we’d have to update all those entries every time we insert or delete something.
2 Implementation using an array

An alternative to using linked lists is to store the data in an array. That should make reading and writing
individual entries nice and easy, since that’s what arrays are really good at. On the other hand, arrays are
not designed to delete elements from the middle or insert them in the middle. Whenever we do so, we will
need to move all the items which occur to the right of that position. Also, arrays don’t dynamically grow
just because we want to insert more stuff. So when — as a result of our insertions — the array is not large
enough any more, we need to allocate a new larger array and copy over all the data.

So internally, we will need three (private or protected) variables:

T *a; // the array that contains all the data.
int length; // the number of elements we are storing right now.
int arraysize; // the size of the array, including elements not currently in use.

Whatever array size we start out with, there may be a time (after enough insertions) when the array is
not large enough. At that time, we will allocate a new larger array, and copy all the items over, then change
a to point to the new larger array (and of course de-allocate the old array to avoid memory leaks). How
much larger should that new array be? There are different approaches:

- Increase the size by 1. This way, we never “waste” space, because the array is exactly large enough to
  hold all its elements. But this is also inefficient, because we have to copy the whole array every time
  a new element is added. And copying is the most expensive part of the operations.

- A natural choice is to double the size of the array every time it becomes too small. That way — except
  for removals — the array is never more than twice as big as it really needs to be, and we are not
  “wasting” much space. And we will see below that we also don’t waste much time on array copying
  this way.

- As a solution somewhere between, we could grow the array by some factor other than 2, say, by
  increasing the size by 10% or something like that.

Now, implementing the get and set functions is really easy (assuming again that we have already written
a function inRange):

T ArrayList::get (int pos) {
    if (inRange(pos)) return a[pos];
}

void ArrayList::set (int pos, const T & newvalue) {
    if (inRange(pos)) a[pos] = newvalue;
}

For the insert function, we’ll want to do something like this first:

void ArrayList::insert (int pos, const T & value) {
    length++;
    if (length > arraysize)
        //allocate new array and copy stuff over
}

Next, we need to make room at position pos by moving all entries from pos to length-2 one position to the
right. Our first thought might be the following:

for (int i=pos; i < length-1; i++)
    a[i+1] = a[i];
However, this doesn’t work because we are overwriting everything with the same initial value. By the time we get to position \( \text{pos+2} \), we have overwritten it with the value from \( \text{pos+1} \), which itself was previously overwritten with the value from \( \text{pos} \). The easiest way to fix this is to move the elements in reverse order.

```java
for (int i=length-2; i >= pos; i--)
    a[i+1] = a[i];
```

Notice that the starting index is \( \text{length-2} \) because we have already incremented \( \text{length} \) earlier, so \( \text{length-1} \) is the last element that should be in use, and we’re copying to position \( \text{i+1} \).

After we have shifted everything to the right to make room at position \( \text{pos} \), we can write the element there using \( \text{a[pos]} = \text{value} \).

The implementation of the \text{remove} function is quite similar. We don’t need to resize the array, and we don’t need to write a new element. But we do need to shift all elements from \( \text{pos+1} \) all the way to \( \text{length-1} \) one position to the left. That is done with a very similar loop. Notice that here, the loop will run from left to right (\( \text{pos+1} \) to \( \text{length-1} \)), because the element in position \( i \) should be saved into position \( i-1 \) before position \( i \) is overwritten.

### 3 Comparison of Running Times

Now that we have seen two implementations, we want to compare them and figure out which one is better. Since they have different advantages for different types of operations, we’ll want to analyze the running time of the different operations in big-\( O \) notation, and see what comes out.

Let’s start with the implementation based on linked lists. Here, all functions start by calling the \text{locate} function. When \text{locate} is called on position \( i \), it takes \( \Theta(i) \) steps, since that’s the number of steps it takes through the list. After that, all of the operations take just constant time to return or overwrite the content of the node at position \( i \), or to update a constant number of pointers. So the running time is \( \Theta(i) \) for all operations, though we also want to keep in mind that it’s all spent just scanning through a list and not overwriting — the amount of writing is \( \Theta(1) \).

For arrays, the \text{get} and \text{set} functions take constant time \( \Theta(1) \), since they know exactly what element to access. The \text{remove} and \text{insert} functions need to move all elements located to the right of the position \( i \). There are \( \Theta(n-i) \) of those elements, so we spend a total of \( \Theta(n-i) \) steps moving things. So in summary, we get the following running times.

<table>
<thead>
<tr>
<th>Function</th>
<th>Linked List</th>
<th>Array</th>
</tr>
</thead>
<tbody>
<tr>
<td>get ( (i) )</td>
<td>( \Theta(i) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>set ( (i,\text{newvalue}) )</td>
<td>( \Theta(i) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>remove ( (i) )</td>
<td>( \Theta(i) )</td>
<td>( \Theta(n-i) )</td>
</tr>
<tr>
<td>insert ( (i,\text{value}) )</td>
<td>( \Theta(i) )</td>
<td>( \Theta(n-i) )</td>
</tr>
</tbody>
</table>

In the case of \text{insert} for an array-based implementation, we have been a little casual. We have not accounted for the cost of copying the entire array when the array needs to grow. That would take \( \Theta(n) \) in addition to the \( \Theta(n-i) \). If we allocate new arrays of size just one larger, this time will be incurred for each \text{insert}, which is quite a lot slower.

But when we always double the array size (or multiply it with a constant fraction, such as increasing by 10%), there is a nice analysis trick we can do. It is true that in the worst case, \text{one insert} operation may take a long time (\( \Theta(n) \)). But if we look at a sequence of many \text{insert} operations, then for every one that takes \( \Theta(n) \), there will be \( n \) operations that will now be fast, because we know that the array has \( n \) empty locations, and won’t double in size until they are full. So on average over all operations, the extra cost of \( \Theta(n) \) can be averaged out over the next (or previous) \( n \) operations, so that the total average cost per operation is actually only \( \Theta(n-i) \). Notice that the average here is taken over a sequence of operations, and not over anything random. This type of analysis, which some people consider more advanced, is called amortized analysis, and it is quite common in analyzing more advanced data structures.
The upshot of the table above is the following: if we expect to be doing mostly get and set operations, there’s no debate that an array-based implementation will be much faster than one based on linked lists. If we expect to do a lot of insert and remove, then we have to trade off \( \Theta(i) \) vs. \( \Theta(n - i) \). There is really not much difference: linked lists do better when we access the first half of the list, while arrays do better for the second half. That doesn’t really guide us, unless we have strong reason to believe that our application really will prefer one half over the other.

The one sense in which linked lists have an advantage is that most of their work only involves scanning over the list, while for arrays, much of the work is copying data around. Writing data tends to take a bit longer than reading, so we would expect the array-based implementation to be a little slower. That seems to be borne out by the experiments done as part of Homework 4.

4 Stacks and Queues

Next lecture, we will look in more detail at these structures, but we can get started. Stacks and queues are quite basic data structures, and their functionality can be implemented to run quite fast. They often serve key (if simple) roles in algorithms and inside computer systems.

A queue is the natural analogue of a line at a store: elements arrive one by one, get in the queue, and are removed from the queue in the order in which they arrived. This makes the order of a queue FIFO (first in, first out). Specifically, the functionality of a queue is the following:

- add an element.
- look at the oldest element.
- remove the oldest element.

We can visualize a queue like a conveyor belt on which we place the elements. One of the main uses of queues (outside of something like simulating a store) is to manage access to shared resources (such as a printer or processor). In fact, printers traditionally have a queue implemented on a server: when users try to print documents, those are added to the queue, and processed in the order they were received when the printer finishes the previous job.

A stack is the natural analogue of piling papers, boxes, or other important things on top of each other. At any time, we can only access the most recently added item, which sits on top. To get at the others, we first need to take care of and remove the more recent ones. This makes a stack LIFO (last in, first out). The functionality of a stack is the following:

- add an element.
- look at the most recently added element.
- remove the most recently added element.

A stack is often drawn in analogy with a physical stack of boxes/papers. Much of a program’s variables (everything we don’t put on the heap) is kept on the program stack. When one function in a program calls another (or itself), all its variables (and state) are saved on the stack, and then the next function starts. We always have to finish the execution of the current function (and remove its variables) before we can access the variables of the other functions (those that called this function). Thus, by explicitly implementing a stack, we can get rid of recursion (which is what really happens in compiling a program).

Stacks are particularly useful when parsing programs or other types of recursively defined expressions. In fact, we will see an easy example in the next class, and a more interesting one as a homework question.