8.0 - Week 8 - Workshop (MA)

Learning Objectives

- Understanding Sorted List ADT.
- Understanding Linked List ADT.

Week 8 Padlet Discussion Board link: https://monashmalaysia.padlet.org/fermi/2022week8

More on List ADTs

Question *Submitted Sep 12th 2022 at 10:11:56 am*

We have already seen how the List ADT can be implemented using an internal array. Today we are going to talk about alternative list representations such as lists implemented with *linked nodes* but also *sorted lists* with arrays. What is your opinion on *why* these may be important?

- ○ come on, because this is what the Week 7 lesson was about!
- ○ who cares?
- ◉ these alternative representations may be more effective in some particular cases.

Sorted Lists with Arrays

Sorted lists are lists whose elements are *kept sorted*. That's an additional **invariant** that we have to respect. The benefit of using sorted lists is that the search for an element can be done much more efficiently if compared to *normal* lists.

Here we are going to implement some sorted list methods. Afterwards, we will compare the performance of the addition of an element and also element search for *normal* lists and *sorted* lists. We will be implementing:

- `_index_to_add()` and `index()` using *binary search*, then use these to implement
- `__setitem__()` and `add()`

For the experiment, we will create a series of sequences of words (like a dictionary) of varying size $10 \times n$ for $n \in \{1, 3, 6, \ldots, 201\}$. For each sequence of words, we will add the words in our list and measure the time spent on addition. Afterwards, we will randomly shuffle the words in the sequence and measure the time spent on searching each word in the list. The two measures will be created for both *normal* lists and *sorted* lists.

The result will be shown in a cactus plot.

As you can observe, *adding* elements to a sorted list is more expensive than to an unsorted list. However, *searching* for an element in a sorted list if much faster than in an unsorted one.

Using sorted lists

Question 1 *Submitted Sep 12th 2022 at 11:01:05 am*

Assuming the complexity of element comparisons is $\mathcal{O}(c)$, if we use binary search to find item positions, then the *worst-case* complexity of adding an element to an array-based sorted list is:

- $\mathcal{O}(c)$
- $\mathcal{O}(cn)$
- **$\mathcal{O}(c \log n + n)$**
- $\mathcal{O}(cn \log cn)$
- None of the above

Question 2 *Submitted Sep 12th 2022 at 11:04:41 am*

Recall our array-based **Set** ADT considered last week.

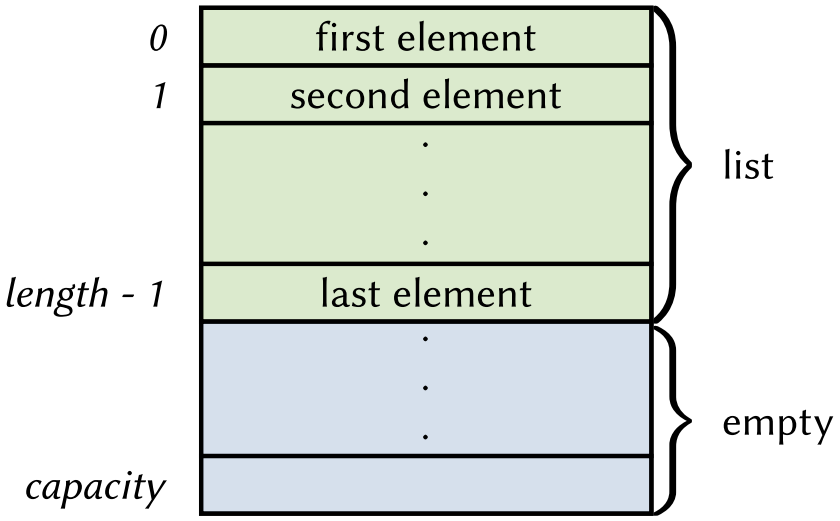
Set union can be computed by (1) adding the items of the first input set and (2) iterating over each item in the second input set, checking if each element was `in` the first input set, and `add`ing it to the result if not. This ends up being $\mathcal{O}(cm(n+m))$, for lists of length m and n, and comparison cost c.

If we implement our sets using sorted collections (arrays or lists) of elements, we can achieve a better complexity of:

- $\mathcal{O}(c)$
- **$\mathcal{O}(c(m+n))$**
- $\mathcal{O}(cm \log(m+n))$
- This is another trick, sorted lists are just as bad.

Array List vs Linked List

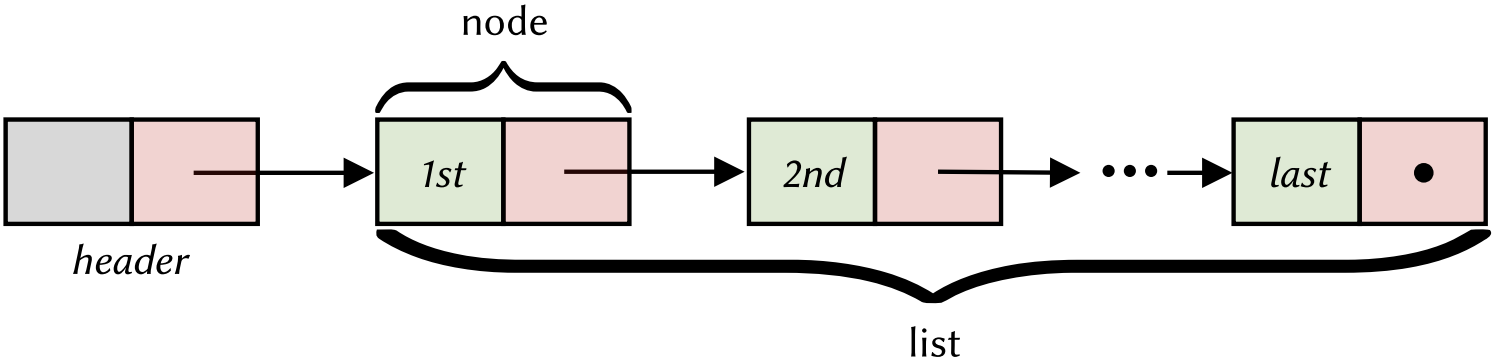
Array-based implementation of the List ADT has the following structure:



Although it *compactly* represents the elements of the list as array stored in one *continuous* chunk of memory, it has a few downsides:

- *inserting* elements to positions different from the end requires shifting (copying!) all the following elements to the right
- *deleting* elements from positions different from the end requires shifting (copying!) all the following elements to the left
- if the list reaches its capacity, a new (larger) chunk of memory should be allocated and the contents of the list should be copied

An alternative implementation of the List ADT using *linked nodes* aims at solving these issues:



An example implementation of the node class is shown here:

```
from typing import TypeVar, Generic
T = TypeVar('T')

class Node(Generic[T]):
```

```
    def __init__(self, item: T = None) -> None:
        self.item = item
        self.next = None
```

Linked List Implementation

In this task, we are going to implement the three modules of the LinkedList class:

- magic method `__setitem__()`
- method `insert()`
- method `delete_at_index()`

Afterwards, we will compare the performance of the *insertion* and *deletion* of an element for both ArrayList and LinkedList. Note that here we will be (1) again dealing with the list of words and (2) inserting and deleting *all the elements* into the **middle of the list**. This will illustrate the benefit of using linked lists.

The result will be shown in a cactus plot.

Feedback Form

Weekly Workshop Feedback Form

Question 1

I am enrolled in:

- □□ Australia
- □□ Malaysia

Question 2

What needs improvement?

No response

Question 3

What worked best?

No response

Question 4

How engaged were you by the workshop?

- □□□ Very engaged
- □□□ Engaged
- □😐□ Not impressed
- 😕😴□ Lost