Abstraction

The first thing that we need to consider when writing programs is the *problem*. However, the problems that we asked to solve in real life are often nebulous or complicated. Thus we need to distill such as system down to its most fundamental parts and describe these parts in a simple, precise language. This process is called **abstraction**. Through abstraction, we can generate a model of the problem, which defines an abstract view of the problem. Normally, the model includes the **data** that are affected and the **operations** that are required to access or modify.

Definition of ADT

An ADT is a mathematical model of a data structure that specifies the type of **data** stored, the **operations** supported on them, and the types of **parameters of the operations**. An ADT specifies what each operation does, but not how it does it. Typically, an ADT can be implemented using one of many different data structures. A useful first step in deciding what data structure to use in a program is to specify an ADT for the program.

Data Structures: ADT and its Implementation

Now let us consider two alternate data structures for the above ADT: (1) **an unordered array** of records and (2) **an ordered array** of records, ordered by IDs. These different data structures greatly influence the implementation details and how fast and efficient the program runs.

First let us look at using an **unordered** array. Assume that the student records are stored in an array with no particular order. We can use a variable *n* to keep track of the number of students currently in the array.

- **ADD**: Simply take the record and store it in slot *n* of the array and increment *n*. This takes constant amount of time since the time is independent of *n*.
- **SEARCH**: Since the array is not ordered, we have to scan through the whole array to find the requested record. The result could be either the record is found or the record doesn't exist. The time taken to perform the search is proportional to *n* and the worst case scenario is *n*. Of course, if the record I am looking for happens to be the fist item in the array, it only takes constant time. However, when determining the running time of an algorithm, we are often interested in worst case analysis.
- **DELETE**: This operation requires us to first search for the given record. Once it is found, the algorithm can simply replace it by the last record and decrement *n*. Once the record is found, it only takes a constant amount of time to delete it. But time spent searching for the record is the same as above, proportional to *n*. Therefore, in all, this operation also takes time proportional to *n*, in the worst case.

Now let us consider using an **ordered** array. Assume that the student records are sorted in an array, with an increasing order of student IDs. We can also use a variable n to keep track of the number of students currently in the array.

- **ADD**: Since the array is sorted, we first need to find out where should we insert the record. This can be done by scanning through the array, comparing the current record in the array with the record we want to insert, and finding the smallest index i of the record whose ID is larger than the new ID. Then the new record should be put to the ith slot in the array. To do that, all the records in slots i, i+1, i+2, ..., n need to be moved down one slot to create an empty slot for the new record. We can then put the new record into the ith slot and increment n. This operation takes time proportional to n.
- **SEARCH**: Since the array of records is sorted by IDs, we can use a **binary search** to find the given record. In general, a binary search algorithm first compares the given ID with the ID in the middle of the array (with index $n/2$ or $(n+1)/2$). Then the algorithm branches based on different conditions: if the two IDs are the same, we find the record; if the given ID is smaller, the algorithm continues to search the first half of the current array, ignoring the second half; otherwise, the algorithm continues to search the second half of the current array, ignoring the first half. The operation time is $\log_2 n$. Finding the time for binary search will be discussed in the future.
- **DELETE**: This operation requires us to first search for the given record. This can be done in $\log_2 n$ time as shown above. Since the array is ordered, we need to fill in the empty slot i with a record, which means the records in slots i+1, i+2, ..., n need to be moved up one slot to cover the empty one. The time is also proportional to n. In all, this operation takes time proportional to n, in the worst case.

A linked list is a linear data structure that includes a series of connected nodes. Here, each node store the **data** and the **address** of the next node. For example,

Linked list Data Structure

You have to start somewhere, so we give the address of the first node a special name called `HEAD`. Also, the last node in the linked list can be identified because its next portion points to `NULL`.

Linked lists can be of multiple types: **singly**, **doubly**, and **circular linked list**. In this article, we will focus on the **singly linked list**.

Representation of Linked List

Let's see how each node of the linked list is represented. Each node consists:

- A data item
- An address of another node

We wrap both the data item and the next node reference in a struct as:

```
struct node
{
  int data;
  struct node *next;
};
```

Understanding the structure of a linked list node is the key to having a grasp on it.

Each struct node has a data item and a pointer to another struct node. Let us create a simple Linked List with three items to understand how this works.

```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data=3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

In just a few steps, we have created a simple linked list with three nodes.

Linked list Representation

The power of a linked list comes from the ability to break the chain and rejoin it. E.g. if you wanted to put an element 4 between 1 and 2, the steps would be:

- Create a new struct node and allocate memory to it.
- Add its data value as 4
- Point its next pointer to the struct node containing 2 as the data value
- Change the next pointer of "1" to the node we just created.

Doing something similar in an array would have required shifting the positions of all the subsequent elements.

In python and Java, the linked list can be implemented using classes as shown in the codes below.

Linked List Utility

Lists are one of the most popular and efficient data structures, with implementation in every programming language like C, C++, Python, Java, and C#.

Apart from that, linked lists are a great way to learn how pointers work. By practicing how to manipulate linked lists, you can prepare yourself to learn more advanced data structures like graphs and trees.

Linked List Implementations in Python, Java, C, and C++ Examples

Python
Java
C
C++

```
// Linked list implementation in C

#include <stdio.h>
```

```c
#include <stdlib.h>

// Creating a node
struct node {
  int value;
  struct node *next;
};

// print the linked list value
void printLinkedlist(struct node *p) {
  while (p != NULL) {
    printf("%d ", p->value);
    p = p->next;
  }
}

int main() {
  // Initialize nodes
  struct node *head;
  struct node *one = NULL;
  struct node *two = NULL;
  struct node *three = NULL;

  // Allocate memory
  one = malloc(sizeof(struct node));
  two = malloc(sizeof(struct node));
  three = malloc(sizeof(struct node));

  // Assign value values
  one->value = 1;
  two->value = 2;
  three->value = 3;

  // Connect nodes
  one->next = two;
  two->next = three;
  three->next = NULL;

  // printing node-value
  head = one;
  printLinkedlist(head);
}
```

Linked List Complexity

Time Complexity

	Worst case	Average Case
Search	O(n)	O(n)
Insert	O(1)	O(1)
Deletion	O(1)	O(1)

Space Complexity: `O(n)`

Linked List Applications

- Dynamic memory allocation
- Implemented in stack and queue
- In **undo** functionality of softwares
- Hash tables, Graphs

Linked lists can be of multiple types: **singly**, **doubly**, and **circular linked list**. In this article, we will focus on the **singly linked list**.

There are three common types of Linked List.

1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Singly Linked List

It is the most common. Each node has data and a pointer to the next node.

Singly linked list

Node is represented as:

```
struct node {
    int data;
    struct node *next;
}
```

A three-member singly linked list can be created as:

```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));
```

```
/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = NULL;

/* Save address of first node in head */
head = one;
```

Doubly Linked List

We add a pointer to the previous node in a doubly-linked list. Thus, we can go in either direction: forward or backward.

Doubly linked list

A node is represented as

```
struct node {
    int data;
    struct node *next;
    struct node *prev;
}
```

A three-member doubly linked list can be created as

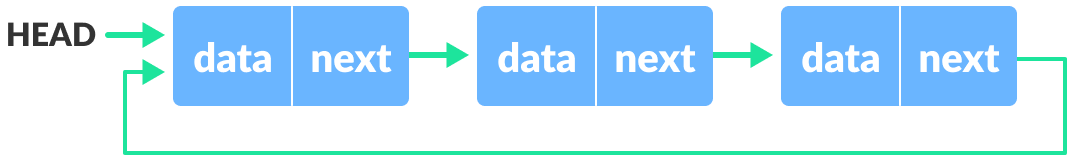
```
/* Initialize nodes */
```

```
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;

/* Connect nodes */
one->next = two;
one->prev = NULL;

two->next = three;
two->prev = one;

three->next = NULL;
three->prev = two;

/* Save address of first node in head */
head = one;
```

If you want to learn more about it, please visit doubly linked list and operations on it.

Circular Linked List

A circular linked list is a variation of a linked list in which the last element is linked to the first element. This forms a circular loop.



Circular linked list

A circular linked list can be either singly linked or doubly linked.

- for singly linked list, next pointer of last item points to the first item
- In the doubly linked list, `prev` pointer of the first item points to the last item as well.

A three-member circular singly linked list can be created as:

```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;

/* Connect nodes */
one->next = two;
two->next = three;
three->next = one;

/* Save address of first node in head */
head = one;
```