Data Structures&Algorithms Week 1

1.1 Intro

Programs = Algorithms + Data Structures

- **Date Structures** efficiently organise data in computer memory
- **Algorithms** manipulate data structures to achieve a given goal

Abstract data types(ADT)

- An **abstract data type(ADT)** is
  - a **type**
    is a collection of values, e.g., integers, Boolean values(true and false)
  - with associate **operations** The operations on ADT might come with mathematically specified constraints, for example on the time complexity of the operations
  - whose representation is hidden to the user
- Example
  - Integers are an abstract data type with operations `+` , `-` , `*` , `mod` , `div` ,...
- List is an ADT; list operations are:
  - insert an entry (on a certain position)
  - delete an entry
  - access data by position
  - search
  - concatenate two lists
  - sort
  - ...
- Different representations of lists
  - Arrays
  - Linked lists
  - Dynamic arrays
  - Unrolled linked lists
- **(Simplified)memory model**
  - Treat memory as a gigantic array
    `Mem[-]`
    # for simplicity we assume that every can contain either an integer, or a string
  - We defined an OS-level pseudocode language, and we call it OS++
    - Two Operations provided by OS++
      - `allocate_memory(n)` - the OS finds a continuous segment of `n` unused locations, designates that memory as used, and returns the address of the first location
      - `free_memory(address,n)` - the OS designates the `n` locations starting from `address` as free

1.2 Arrays

List as an array of a fixed length

Java:

```
final int[] nums = new int[4];
```

OS++:

```
final nums = allocate_memory(4*1);
```

More complicated arrays in *Mem[-]*

Java:

```
class Student {
    String name;
    int id;
}
final Student[] students = new Student[3];
```

OS++:

```
final students = allocate_memory(3*2);
```

Memory management

In java

- memory allocation is **automatic**
- freeing memory is **automatic** (by the garbage collector)
- bounds of arrays **are checked**

In C or C++

- allocations are **explicit** (similar to OS++ and Mem[-])
- freeing memory is **explicit**(similar to OS++ and Mem[-])
- bounds **are not checked**

Java is slower and safe, C(or C++) is fast and dangerous

Inserting by shifting in OS++

To insert a student at position `pos` , where 0 ≤ pos ≤ size:

```
final Student[] students = new Students[maxsize];
int size = 0;

void insert(int pos, String name, int id){
    if (size == maxsize){
        throw new ArrayFullException("Students_array");
    }
    for (int i = size - 1; i ≥ pos; i--){
        Mem[student + 2*i + 2] = Mem[student + 2*i];
        Mem[student + 2*i + 3] = Mem[student + 2*i + 1];
    }
    Mem[student + 2*pos] = name;
    Mem[student + 2*pos + 1] = id;
    size++;
}
```

1.3 Linked Lists

Linked Lists in Mem[-]

The first location of such block stores a number and the second location stores the address of the following block

The constant `list` is an address pointing to the first node, called the head pointer

`END` indicates the end of the list (graphically as □ with x inside)

A linked list is empty whenever `Mem[list]` is equal to `END`

Deleting at the beginning

```
Boolean is_empty(int list) {
    return (Mem[list]) == END);
}

void delet_begin(int list){
    if is_empty(list) {
        throw new EmptyListException("delete_begin");
    }
    final firstnode = Mem[list];
    Mem[list] = Mem[firstnode + 1];
    freememory(firstnode,2);
}
```

Lookup

```
int value_at(int list, int index){
    int i = 0;
    int cursor = list;
    int nextnode = Mem[cursor];
    while (true) {
        if (nextnode == END) {
            throw OutOfBoundsException;
        }
        if (i == index) {
            break;
        }
        cursor = nextnode + 1;
        nextnode = Mem[cursor];
        i++;
    }
    return Mem [nextnode];
}
```

Insert at the end

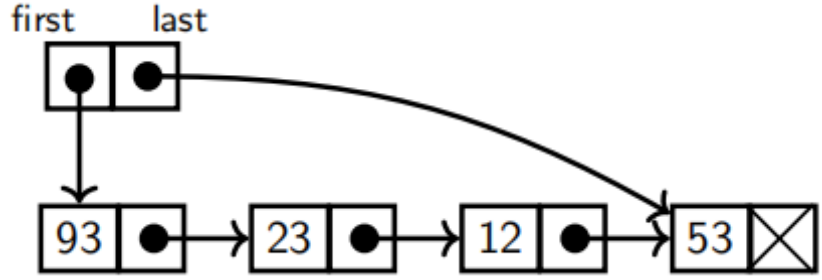
```
void insert_end(int list, int number){
    final newblock = allocate_memory(2);
    Mem[newblock] = number;
    Mem[newblock + 1] = END;

    int cursor = list;
    while (Mem[cursor] != END){
        cursor = Mem[cursor] + 1;
    }
    Mem[cursor] = newblock;
}
```

Comparison

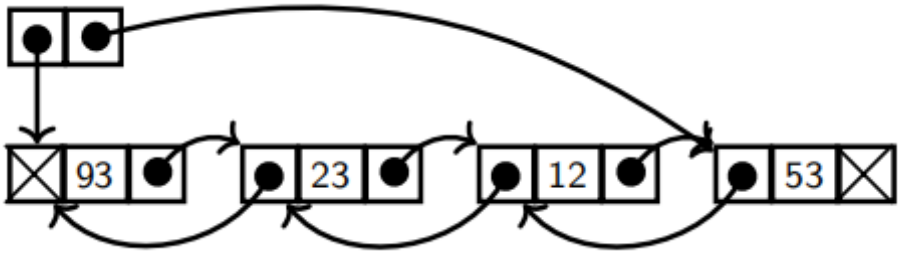
	Array	Linked List
access data by position search for an element	constant linear	linear linear
insert an entry at the beginning inset an entry at the end insert an entry(on a certain position)	linear linear linear	constant linear linear
delete first entry	linear	constant
delete entry `i`	linear	linear
concatenate two lists	linear	linear

Modifications

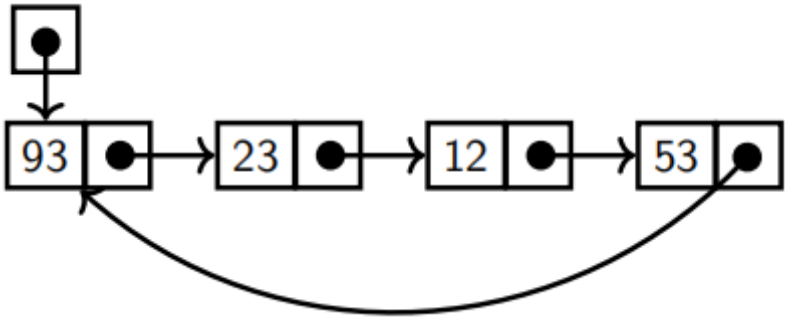
Linked list with a pointer to the last node:



Doubly linked list:



Circular Singly Linked List:



Circular Doubly Linked List: