Lecture 3 : LINKED LISTS.

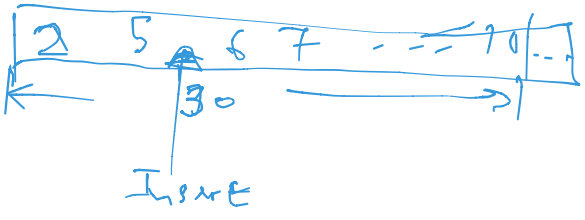
Lists : Store a list of items → Array

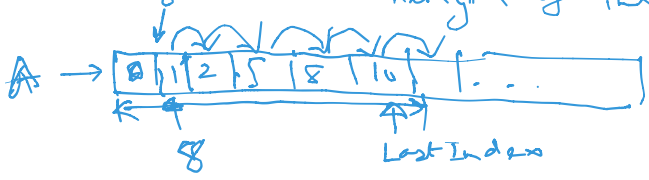
Dynamic alloc.

Disadvantages :

* Arrays have a fixed length.
* Insert an item at the beginning or middle...
  → Time proportional to the length of the array.

int A[50];

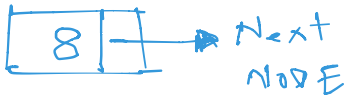




Linked Lists → A recursive ~~data~~ structure

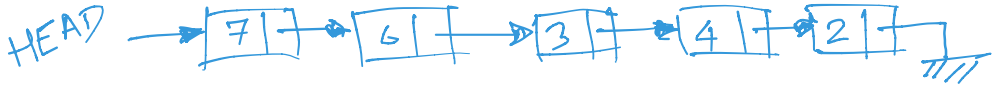
↳ Self-referential

↳ Made up of "NODES"



Each node has two things :

* an item → ~~Array~~ type
* a pointer to the NEXT NODE in the list



~~struct~~ ~~class~~ SLLNode {

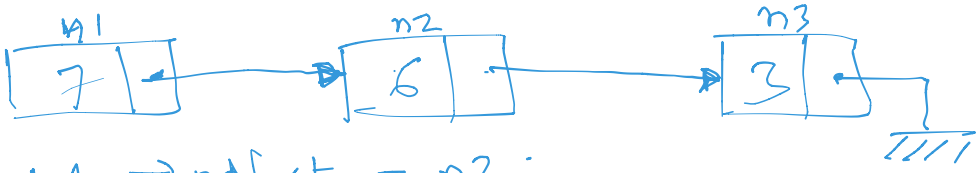
```
int mItem;
SLLNode* mNext;
```

};

SLL
↓
singly linked list


```
SLLNode* n1 = new SLLNode();
n1->mItem = 7;
SLLNode* n2 = new SLLNode();
n2->mItem = 6;
```

SLLNode * n3 = new SLLNode();
n3 -> mItem = 3;



n1 -> mNext = n2;

n2 -> mNext = n3;

n3 -> mNext = nullptr;

```
class SLLNode {
public:
    SLLNode(int aItem, SLLNode * aNext)
    : mItem(aItem), mNext(aNext)
    {}

    SLLNode(int aItem)
    : mItem(aItem), mNext(nullptr)
    {}
...
};
```

Initializer list

7 → 6 → 3 ⊣

```
SLLNode * n1 = new SLLNode(7, new SLLNode(6,
                           new SLLNode(3)));
```

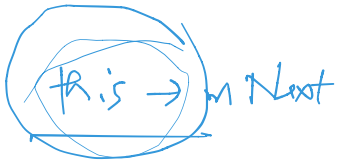
Advantages of Linked lists over arrays

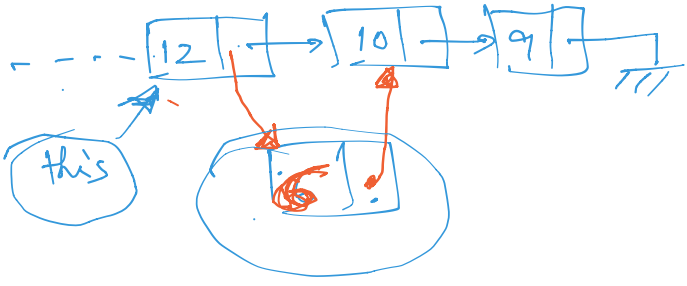
* If you have a pointer to a previous node, inserting an item into the middle of a linked list takes constant time.
* List can keep growing until memory runs out,

⇒ No need to resize the list.

Inserting a new item after "this" node

```
class SLLNode {
public:
void insertAfter(int aItem)
{
mNext = new SLLNode(aItem, mNext);
}
};
```

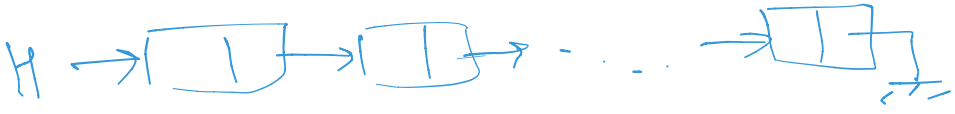




Disadvantages:
(of linked lists)

* Finding the n-th item in the list takes time proportional to the length of the list.

499th



A[499-1]

Tension between inserting a new item Vs finding a given item.

List of objects

Item → Pointer to any object

class SLLNode {

```
    Item* mItem;
    SLLNode* mNext;
};
```

A SLL (Singly Linked List) class

problems w/ SLLNode:

(1) Data can easily get corrupted.

(2) Empty list.

Encapsulation

Separate SLL class : Maintain the Head pointer,

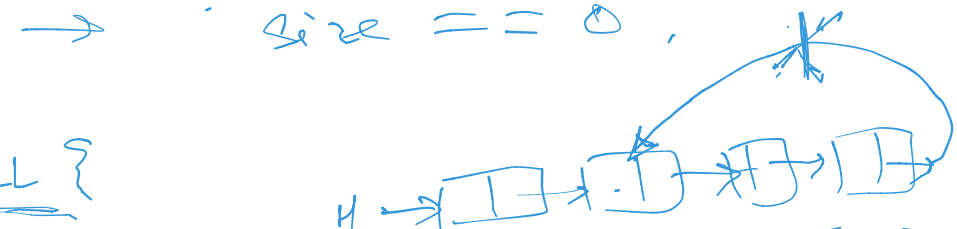
Size of the list. → size.

⇒ Invariant associated w/ the list

Empty list → size == 0.

ADT

```
class SLL {
public:
    void insertItem(int aItem) { ... };
private:
    SLLNode* mHead;
    int mSize;
};
```



this

Why make private?

(1) protection against corruption

A private member is invisible and inaccessible to other classes.

(2) You can improve the implementation without causing other classes to fail.

Interface of a class: prototype for the public member function, and description of its behavior.

Abstract Data Type (ADT): a class with a well-defined interface, but the implementation details are hidden from other classes. → flexibly implement in different ways.

Invariant: a fact about a data structure that is always true.

There are classes which are not ADTs;
Sometimes the objective of a class is to only store data. → No invariant.

SLL ADT: Enforces * invariants.

* mSize is always correct.
* List is never circularly linked.

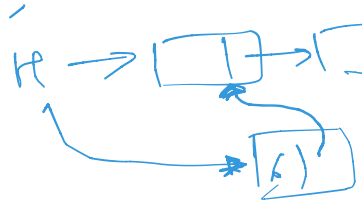
→ ONLY SLL methods are allowed to change the list.

* Members are private.
* No method in the SLL should return (SLLNode *).

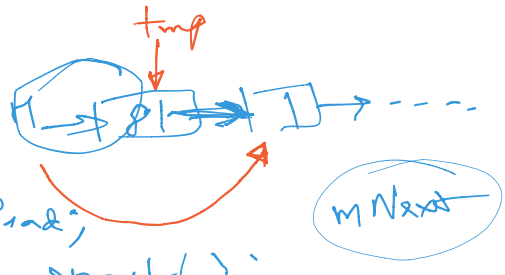
Doubly-linked lists (DLL)

Insert / Delete an item in the front of a SLL → Easy

```
class SLL {
public:
  void insertFront(int anItem) {
    mHead = new SLLNode(anItem, mHead);
    mSize++;
  }
```



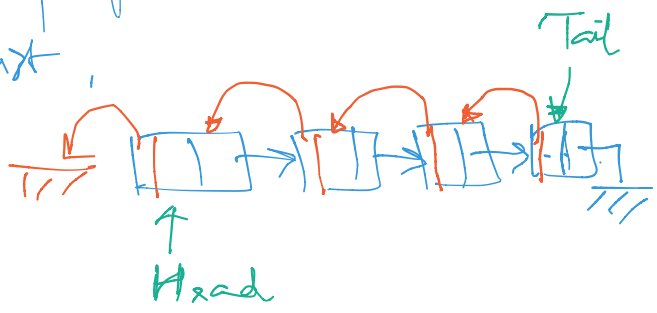

```
  void deleteFront() {
    if (mHead != nullptr) {
      SLLNode * tmp = mHead;
      mHead = mHead->next();
      delete tmp;
      mSize--;
    }
  }
};
```



Insert / Delete at the end of the list

→ take time proportional to

the size of the list.



→ DLL

```
class DLLNode {
    ItemType  mItem;
    DLLNode*  mNext;
    DLLNode*  mPrev;
};

class DLL {
    DLLNode*  mHead;
    DLLNode*  mTail;
};
```

Space × Time

Subclass

⇒ Insert and delete items at both ends in constant running time.

More elegant DLL:

At least 2 items in the DLL:
mHead, mTail

Sentinel Node: A special Node that does not represent an item.

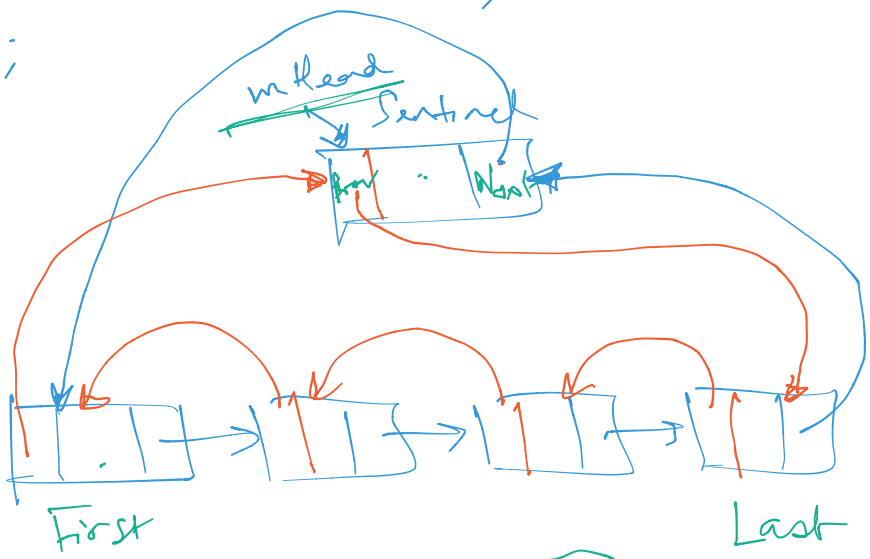
Circularly Doubly linked list →

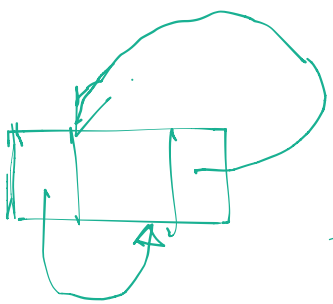
CDLL

mHead → Sentinel.

(Does not store an item)

```
class CDLL {
    DLLNode * mHead; // Sentinel
    int mSize;
};
```





Empty

⇓

Prev and next point to itself.