What is data structure?

A data structure is a way to store and organize data in a computer, so that it can be used efficiently.

Why data structure?

They are essential ingredients in creating fast and powerful algorithms

They help to manage the organize data.

They make code cleaner and easier to understand.

Complexity analysis:

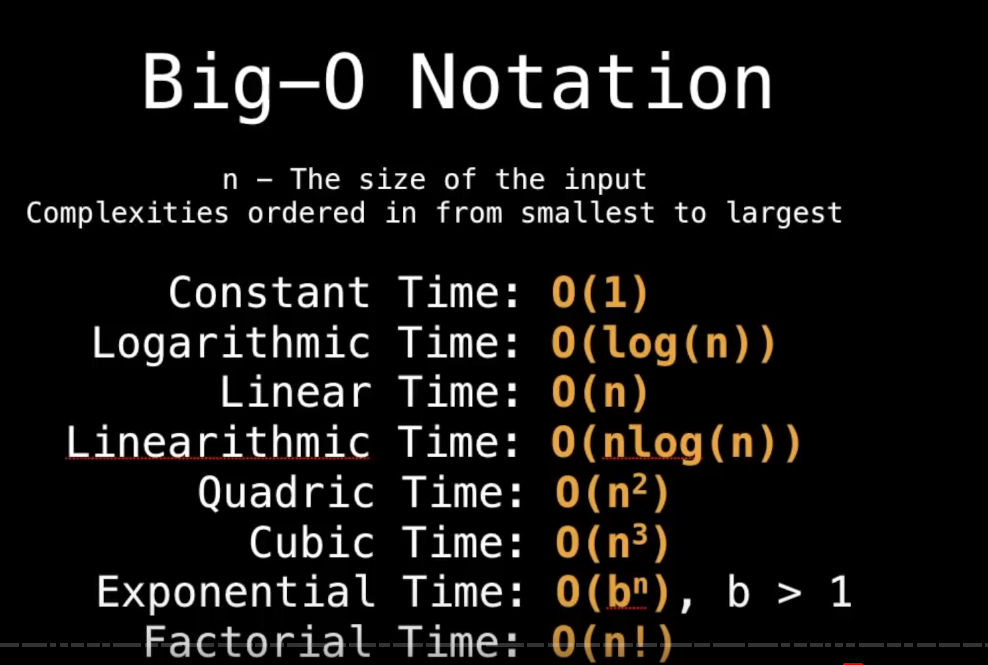
How much time does this algorithm need to finish?

How much space does this algorithm need to finish?

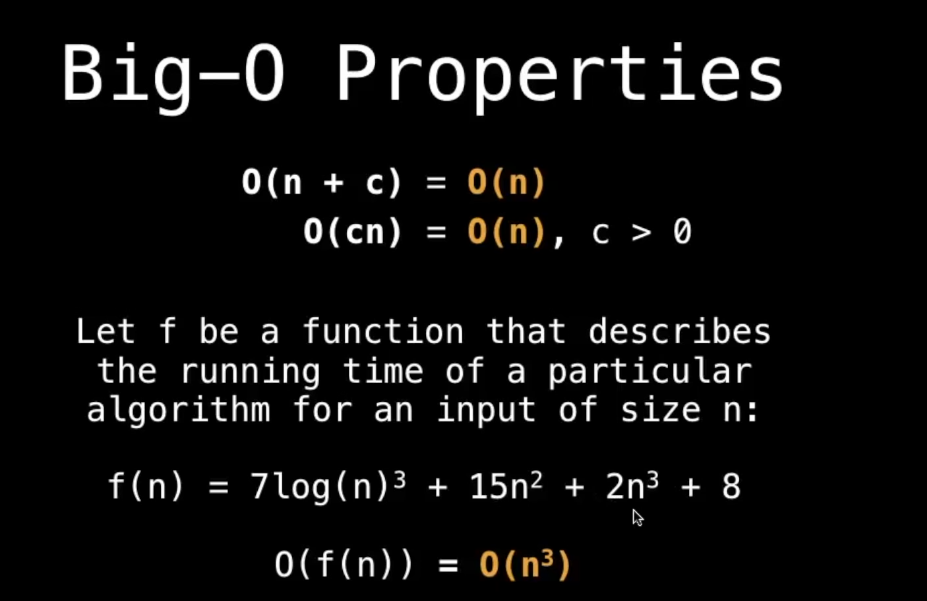
Big O only cares about the worst case.

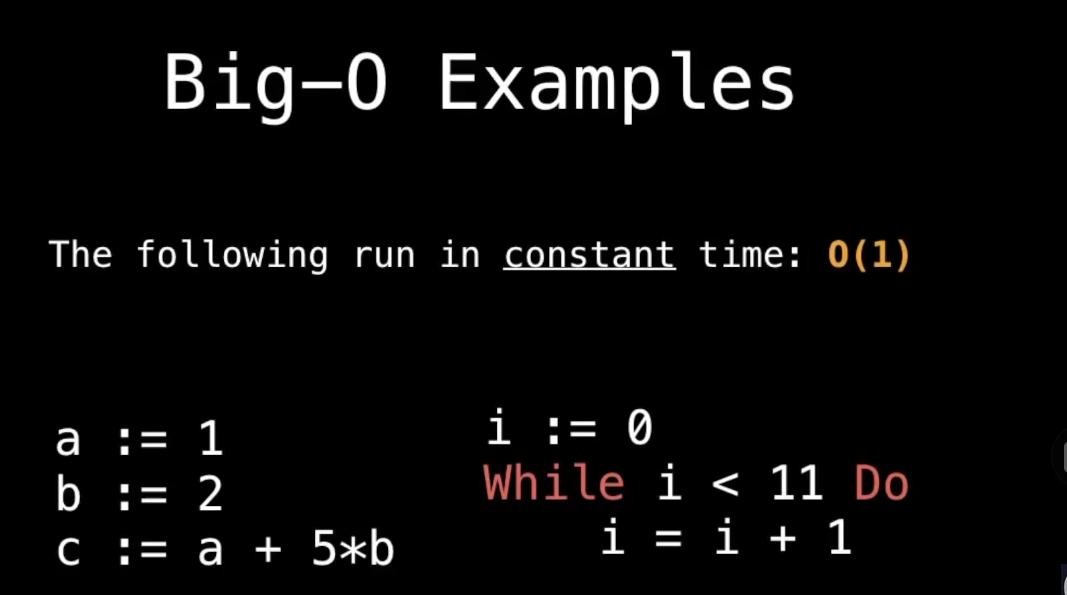
Big O notation gives an upper bound of the complexity in the worst case, helping to quantify performance as the input size becomes arbitrarily large.

N as the input size

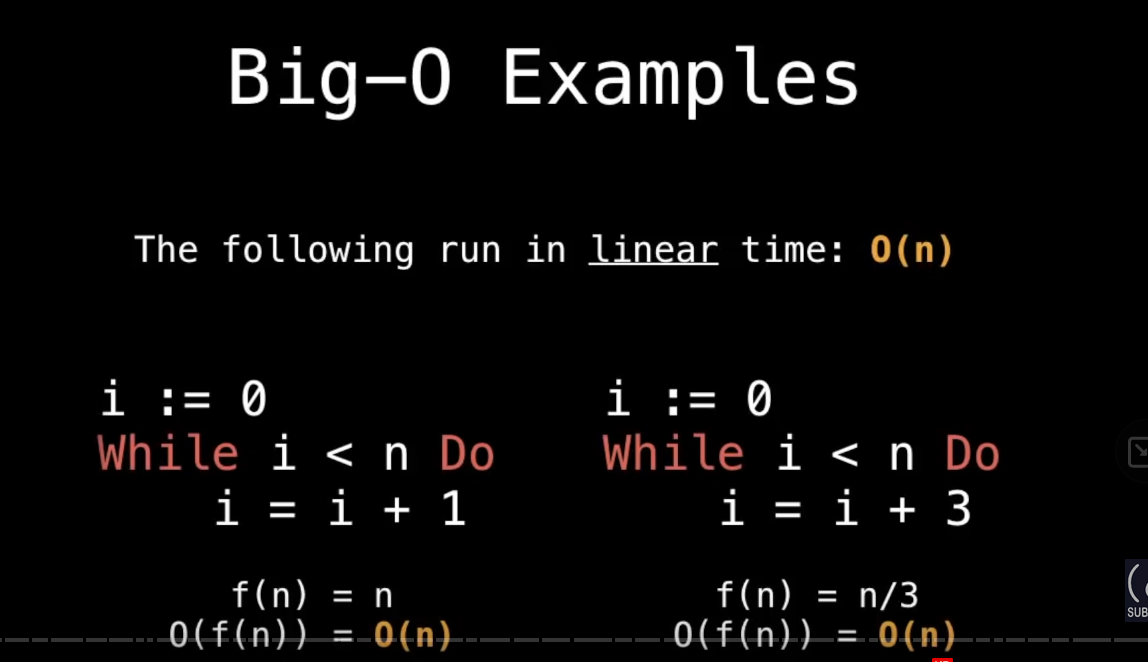


Big O properties:

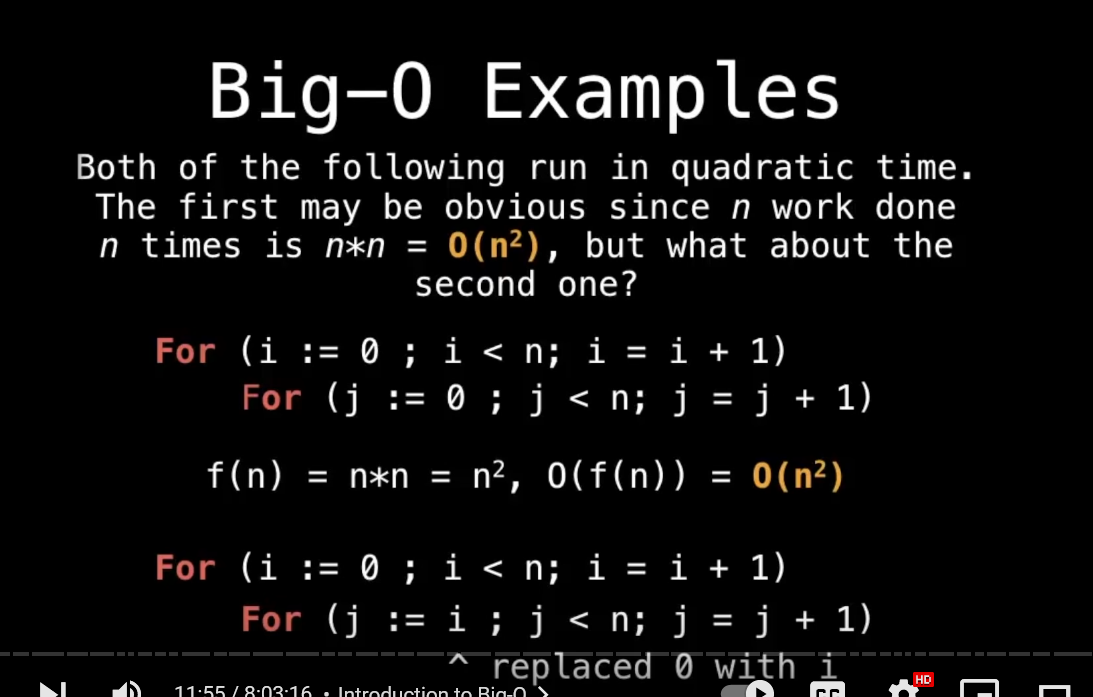
Big O only cares about the biggest number( or most dominant) :  
 

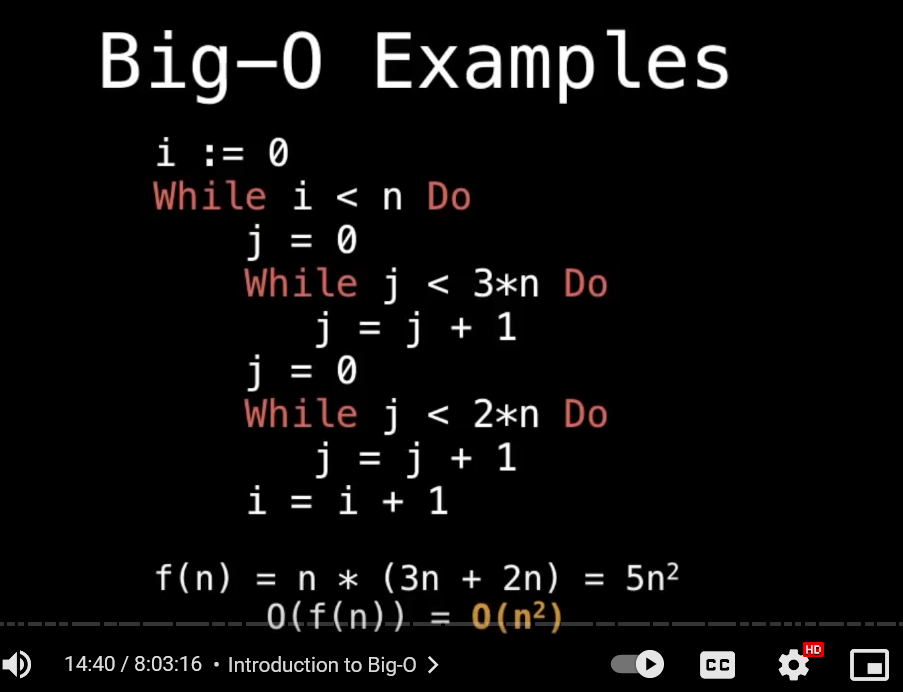
Practice:

Why? Because the when it is execution times is a constant. The complexity of big O is 1.



Big-O quadratic:





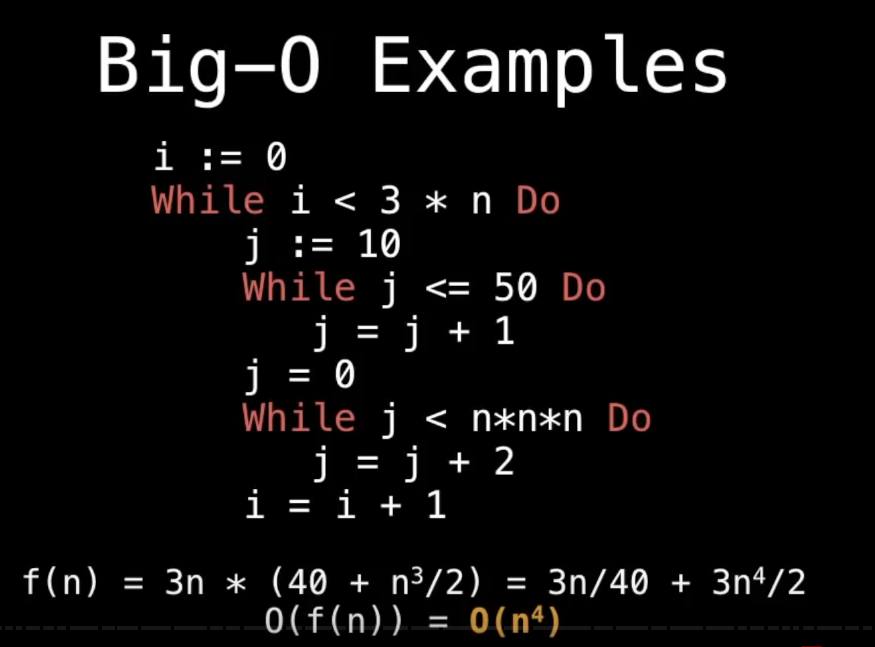
As you can see, if there is a outer loop.

The steps from the outer N \* (inner steps)

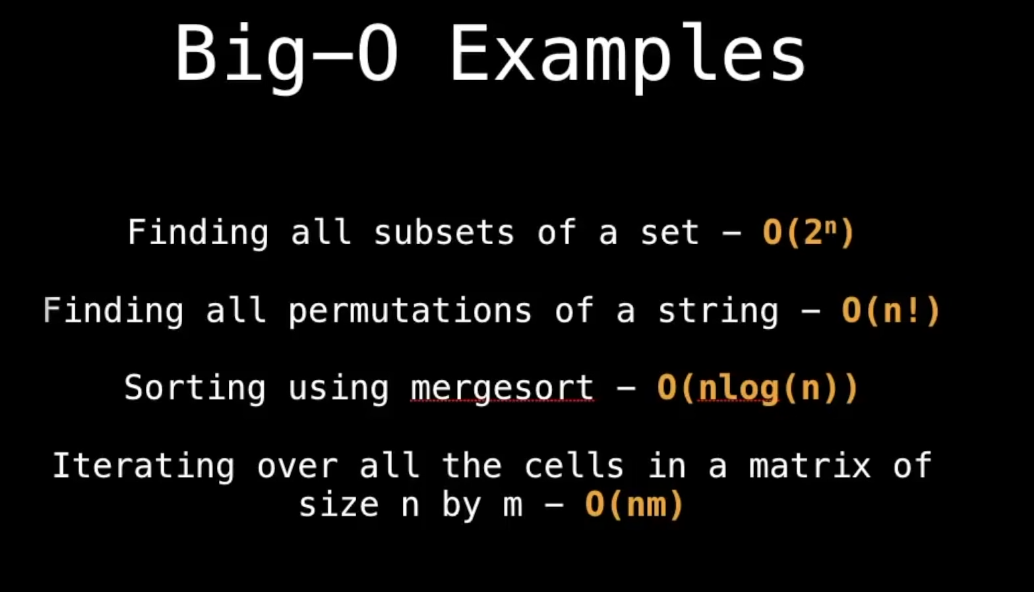
So, we have, N\* (3n+2n) = 5n\*n

But, however, remember big O only cares about biggest part

So it is N\*N



Notice inside the second inner loop, j+2 will cause the complexity divided by 2.



# **List Stack and queues**

# ADT?

An abstract data type is an abstraction of a data structure which provides only the interface to which a data structure must adhere to.

Lists, sets, and graphs can be viewed as ADTs.

There are no rules telling use which operations must be supported for each ADT. ( depends on the design of yours)

The interface does not give any specific details about how something should be implemented or in what programming language.

# The list ADT:

Size of the list is N, when N=0, this list is an empty list.

From A(0) to A(N-1)

For any list except the empty list, we say that

Ai follows Ai-1(i<n) sequence. Ai-1(i>0) precedes Ai.

Any position before A0 and AN-1 is undefined.

The position of element Ai in a list is i.

Some operations are printList, makeEmpty, find, insert, findKth, remove, next, previous.

List and array:

When and where is a static Array used?

Storing and accessing sequential data

Temp objects.

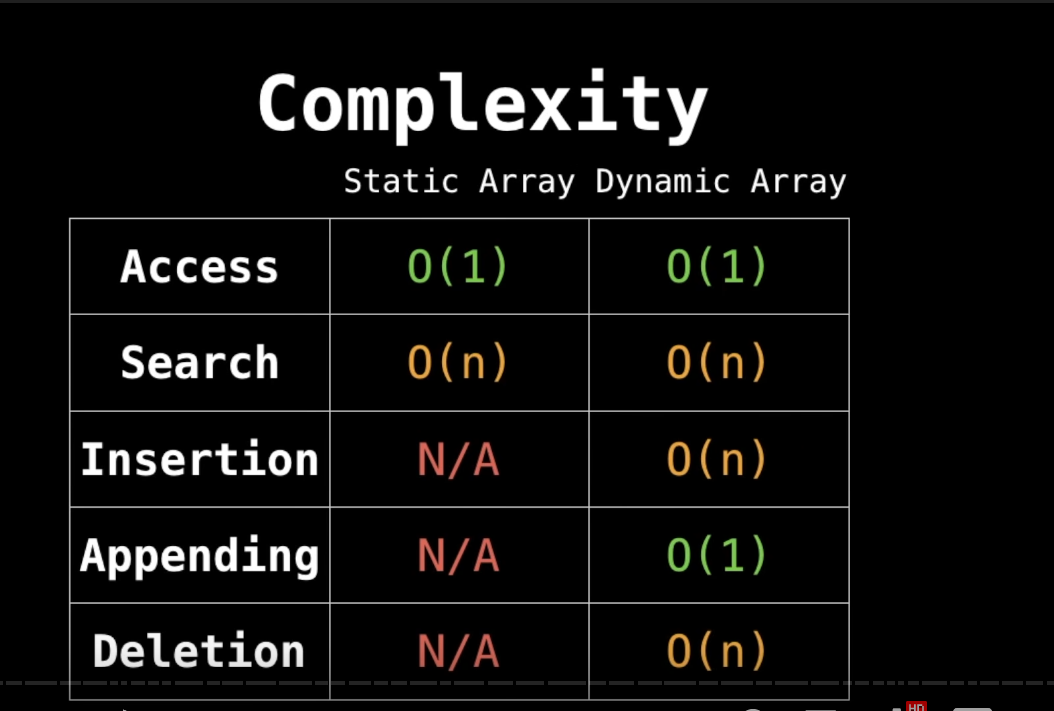
Used by IO routines as buffers.

Look up tables and inverse loop up tables

Return multiple values from a function

Used in dynamic programming to cache answers to subproblems

Complexity:



Simple linked list:

As you can see, deletion and insertions on an array is quite expensive.

So, that’s why we need the linked-list.

Linked list contains a series of nodes, which are not necessarily adjacent in memory.

Each node contains the element and a link to a node containing its successor. We call this NEXT link.

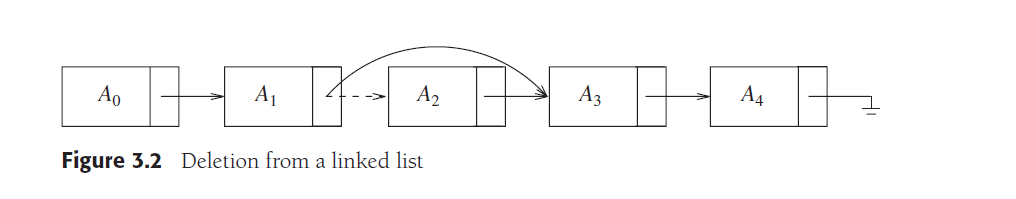
The last cell’s NEXT pointing at nullptr.

Implement the operations on the linked list:

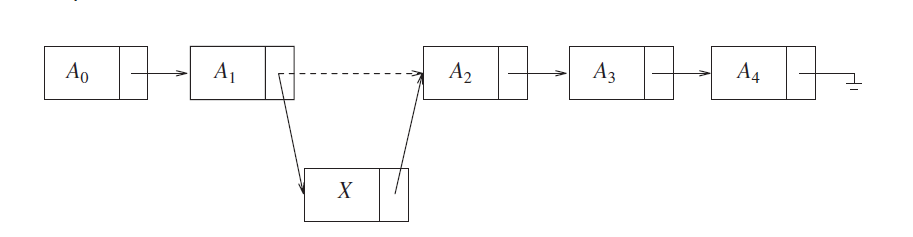
Some operations like find(x) or printList() we have to start at the first node, and transverse the list by following the next links. O(n)

The findKth(index) is definitely less efficient than array.

( first takes O(i) time to reach the position, and then tranversing down)

Remove() method can be executed in one next pointer change. 

Insert() method requires obtaining a new node from the system by using new call.



Then executing **two next pointer** maneuvers. (the new’s next points to the previous pointing ,the precious’s next point to the new)

Special case adding or removing at the front:

Adding end is easy:

O(1)

As long as we maintain a link to the last node. ( like tail )

Removing end is not that easy:

1. Find the next to last item
2. Change the next link to nullptr
3. And then update the link maintains the last node.

This is even harder when the node only has next link.

So Doubly linked list is introduced.

Term?

Head: The first node in a linked list.

Tail: The last node in a linked list

PTR: references to another node.

Node: An object containing data and pointers.

# STL vector and list

List ADT is implemented in STL. Based List ADT, they implemented queues and stacks, and more.

(page 80)

Std::Vector is implementation of List ADT??

Cheap on accessing.

Std::list

Cheap on insertion of new items and removal of existing items.

But not indexable, so you need the iterator to transverse.

Be aware std::list is doubly linked list. ( not singly linked list )

Common Methods: ( supported by all containers )

Int size()

Void clear()

Bool empty() const

These three methods are supporting adding and removing from the end of the list in constant time. O(1)

1. void push\_back( const Object & x ): adds x to the end of the list.
2. void pop\_back( ): removes the object at the end of the list.
3. const Object & back( ) const: returns the object at the end of the list (a mutator that

returns a reference is also provided).

1. const Object & front( ) const: returns the object at the front of the list (a mutator that

returns a reference is also provided).

Accessing the head or tail in const time.

Because a doubly linked list allows efficient changes at the front, but a vector does not, so the following methods are only supported by list.

void push\_front( const Object & x ): adds x to the front of the list.

void pop\_front( ): removes the object at the front of the list.

Also, vector has its own methods that list does not have:

1. Object & operator[] ( int idx ): returns the object at index idx in the vector, with no

bounds-checking (an accessor that returns a constant reference is also provided).

1. Object & at( int idx ): returns the object at index idx in the vector, with boundschecking

(an accessor that returns a constant reference is also provided).

1. int capacity( ) const: returns the internal capacity of the vector. (See Section 3.4 for

more details.)

1. void reserve( int newCapacity ): sets the new capacity. If a good estimate is available,

it can be used to avoid expansion of the vector. (See Section 3.4 for more details.)

random accessing is pretty sweet. And also, it only cost const of time.

# Iterators:

In STL, a position is represented by a nested type=====🡺Iterator.

Iterator is always pointer-wise higher than the elements stored inside the container.

Syntax:

Your typeName is:

List<typeName>;

Your iterator is:

list<typeName> ::iterator

All the STL containers have following methods:

iterator begin( ): returns an appropriate iterator representing the first item in the

container.

iterator end( ): returns an appropriate iterator representing the endmarker in the

container (i.e., the position after the last item in the container). ( like slicing in python)

So this is a typical for loop implemented for a STL container:

for( vector<int>::iterator itr = v.begin( ); itr != v.end( ); ++itr)

cout << \*itr << endl;

Iterator methods:

1. itr++ and ++itr: advances the iterator itr to the next location. Both the prefix and

postfix forms are allowable.

1. \*itr: returns a reference to the object stored at iterator itr’s location. The reference returned may or may not be modifiable (we discuss these details shortly).
2. itr1==itr2: returns true if iterators itr1 and itr2 refer to the same location and false

otherwise.

1. itr1!=itr2: returns true if iterators itr1 and itr2 refer to a different location and false otherwise.

Containers also have methods that require iterators:

1. iterator insert( iterator pos, const Object & x ): adds x into the list, prior to the

position given by the iterator pos. This is a constant-time operation for list, but not for

vector. The return value is an iterator representing the position of the inserted item.

1. iterator erase( iterator pos ): removes the object at the position given by the iterator.

This is a constant-time operation for list, but not for vector. The return value is

the position of the element that followed pos prior to the call. This operation invalidates

pos, which is now stale, since the container item it was viewing has been removed.

1. iterator erase( iterator start, iterator end ): removes all items beginning at position

start, up to, but not including end. Observe that the entire list can be erased by

the call c.erase( c.begin( ), c.end( ) ).

Example of using erase on a list:

When removing **every second** element in a list, **using erase method would be a linear-time O(n) because each calls to erase takes constant time, but in a vector the entire routine will take quadratic time.**

*1* template <typename Container>

*2* void removeEveryOtherItem( Container & lst )

*3* {

*4* auto itr = lst.begin( ); // itr is a Container::iterator

*5*

*6* while( itr != lst.end( ) )

*7* {

*8* itr = lst.erase( itr );

*9* if( itr != lst.end( ) )

*10* ++itr;

*11* }

*12* }

Don’t use erase on vector then when you have to remove every second element.

Const\_iterators:

\*iterator is not just the value of the item inside the container, but also the element itself.

This is to protect the data from changing it directly and bypass all the validation.

Std::vector implementation:

Vector can be copied and memory is handled interally.

Array:

Simply a pointer variable to a block of memory.

The actual array size must be maintained separately by the programmer.

The block of memory cannot be resized, but new presumably larger block can be obtained and initialized with the old block, and then the old block can be freed.

Vector:

will maintain the primitive array ( via a pointer variable to the block of allocated memory), the current number of items stored in the vector.

Implemented rule of 5. Operator = is overloaded.

Will provide a resize routine that will change the size of the vector and a reserve routine that will change the capacity of the vector.

Vector will provide an implementation of subscript operator. (accessor and mutator)

Vector implemented all the basic routines.

Vector supports all the iterators and const\_iterator.

However, like all the STL containers, vector is limited error checking.

Std::list implementation:

The list class itself, contains begin, end, size, and a set of methods.

The node class, which is likely to be a private nested class. Contains the data and pointers to the previous and next node.

The const\_iterator class which abstracts the notion of a position, and is a public nested class.

The iterator class, abstracts the notion of a position, and is a public nested class.

The iterator classes store a pointer to the current node, and the end marker is a valid position.

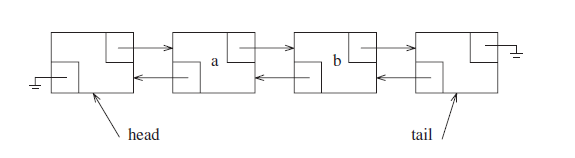
We could add either from the front or from the end.

Node beginning is called header, node end is called end.

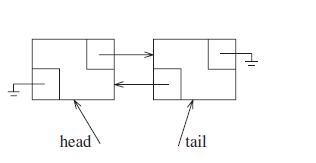
There is an extra node at the front of the list, logically representing the beginning marker.

--- So, these extra nodes are known as the sentinel nodes.

The advantage of using the extra nodes is that they greatly simplify the coding by removing a host of special cases.



An typical doubly linked list without elements but only the head and tail.

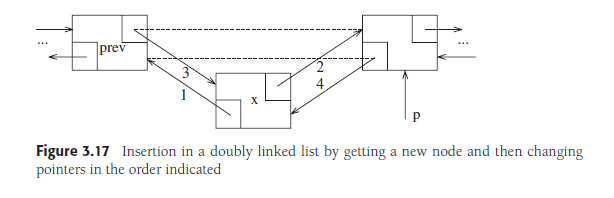


List implementation in C++:

Please find the code from the example code.

Important implementation detail:

Insertion in a doubly linked list



Step 1 and 2

// creating a new node, let the next pointed at p(nextNode), let the prev pointed at the prev( previous node)//

Node\* newNode = new Node{x,p->prev,p} step 1 and 2

// Reset the prev and p ‘s next and prev respectively

p-> prev = p->prev->next = newNode;

letting

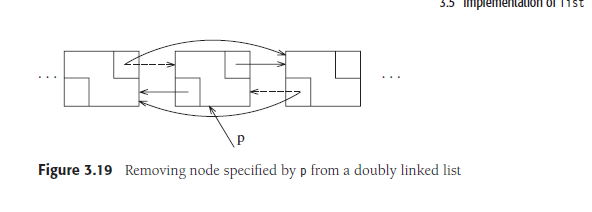
p’s prev is prevNode, and its next is pointed at the newNode Now

Then finally you can change the p’s prev to the newNode

p-> prev pointed at newNode,

The code is so concise that it is slightly unreadable at first.

Erase in a doubly linked list:

Removing node specified by p from a doubly linked list:  
 

Just like insert, erase must update theSize.

Pretty much you let the

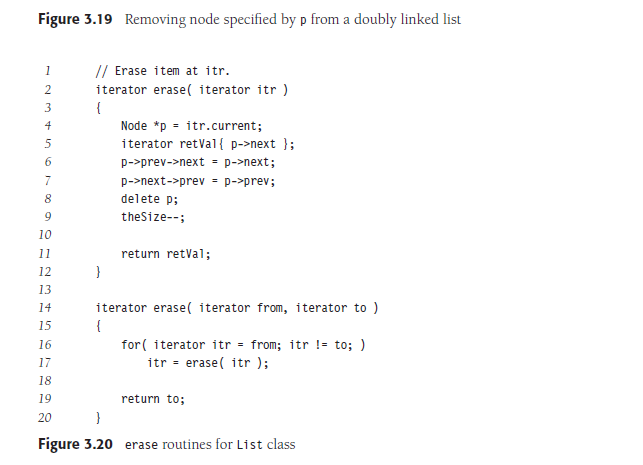
P’s prev’s next pointed at p’s next

P’s next’s prev pointed at p’s prev.

Then remove.

Be aware you need a temp remover to pointed at the p (so no memory lost)

Then update theSize

Then return the current iterator position. 

Array vs linked list:  
 Before you decide which structure you use, you must identify which operation would you use most frequent.

1. Accessing an element:

Array: using offset between the first and the index you want to reach, so the O(1)

Linked List: You will have to go through every single node in the worst case. So, the O(n)

1. Memory requirements:

Array: fixed size.

Linked list: No unused size, extra memory for nodePTR.

Normally depends on the size of the data type stored inside this kind of container.

If the size of the data is big, list is for sure preferred.

1. Cost of inserting an element

At beginning

Array O(n) list O(1)

At end

Array O(1) (not full) list O(n)

Array O(n) full

At random position:

Array O(n) list O(n)

Singly linked list only hold a reference to the next node.

In the implementation you always maintain a reference to the head to the linked list and a reference to the tail node for additions / removals.

With a doubly linked list, each node hold a reference to the next and previous node. In the implementation, you always maintain a reference to the head and the tail of the doubly linked list to do quick additions removals from both ends of your list.

Pros and Cons:

