**Data structure Using Python**

**Section 1:**

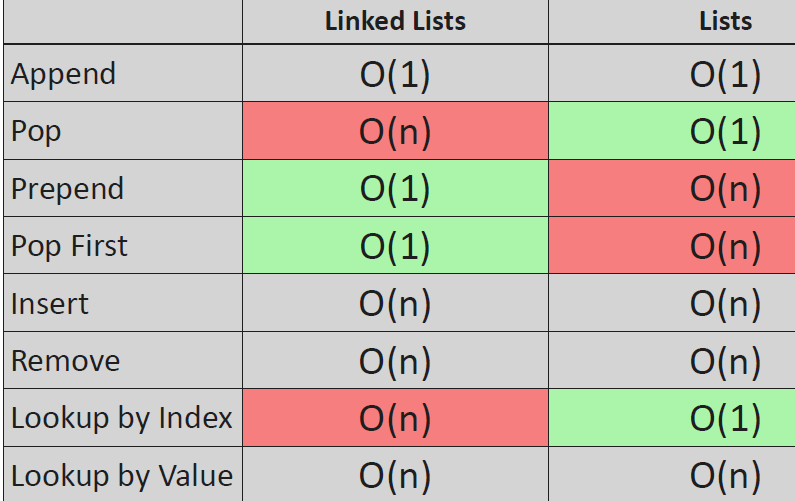
* setting up the environment
* differences between data structures and abstract data types

**Section 2 - Arrays:**

* what is an array data structure?
* arrays related interview questions

**Section 3 - Linked Lists:**

* linked list data structure and its implementation
* doubly linked lists
* linked lists related interview questions



**Section 4 - Stacks and Queues:**

* stacks and queues
* stack memory and heap memory
* how the stack memory works exactly?
* stacks and queues related interview questions

**Section 5 - Binary Search Trees:**

* what are binary search trees
* practical applications of binary search trees
* problems with binary trees

**Section 6 - Balanced Binary Trees (AVL Trees and Red-Black Trees):**

* why to use balanced binary search trees
* AVL trees
* red-black trees

**Section 7 - Priority Queues and Heaps:**

* what are priority queues
* what are heaps
* heapsort algorithm overview

**Section 8 - Hashing and Dictionaries:**

* associative arrays and dictionaries
* how to achieve **O(1)** constant running time with hashing

**Section 9 - Graph Traversal:**

* basic graph algorithms
* breadth-first
* depth-first search
* stack memory visualization for DFS

**Section 10 - Shortest Path problems (Dijkstra's and Bellman-Ford Algorithms):**

* shortest path algorithms
* Dijkstra's algorithm
* Bellman-Ford algorithm
* how to detect arbitrage opportunities on the FOREX?

**Section 11 - Spanning Trees (Kruskal's and Prim's Approaches):**

* what are spanning trees
* what is the union-find data structure and how to use it?
* Kruskal's algorithm theory and implementation as well
* Prim's algorithm

**Section 12 - Substring Search Algorithms**

* what are substring search algorithms and why are they important in real world software’s
* brute-force substring search algorithm
* hashing and Rabin-Karp method
* Knuth-Morris-Pratt substring search algorithm
* Z substring search algorithm (Z algorithm)
* implementations in Python

**Section 13 - Hamiltonian Cycles (Travelling Salesman Problem)**

* Hamiltonian cycles in graphs
* what is the travelling salesman problem?
* how to use backtracking to solve the problem
* meta-heuristic approaches to boost algorithms

**Section 14 - Sorting Algorithms**

* sorting algorithms
* bubble sort, selection sort and insertion sort
* quicksort and merge sort
* non-comparison based sorting algorithms
* counting sort and radix sort

In the first part of the course we are going to learn about basic**data structures such** as linked lists, stacks, queues, binary search trees, heaps, and some advanced ones such as AVL trees and red-black trees.. The second part will be about graph algorithms such as spanning trees, shortest path algorithms and graph traversing. We will try to optimize each data structure as much as possible.

Most of the advanced algorithms relies heavily on these topics so it is definitely worth understanding the basics. These principles can be used in several fields: in investment banking, artificial intelligence, or electronic trading algorithms on the stock market. Research institutes use Python as a programming language in the main: there are a lot of library available for the public from machine learning to complex networks.

**\*\*\* Data Structures \*\*\***

what are data structures?

“Data structures came to be in order to manage large amounts of data efficiently for uses such as large databases and internet indexing services”

So modern applications and software’s are manipulating a huge amount of data.

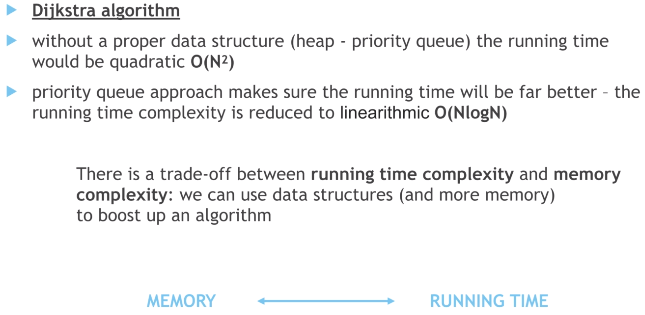
Ex: Facebook, Instagram or Twitter, all of these applications and software’s have to deal with a huge amount of data.

* How to deal with these huge amounts of data, how to make sure that the operations are going to be fast enough.
* Somehow we have to make sure the application is as fast as possible. And this is exactly why using the right data structures is crucial.

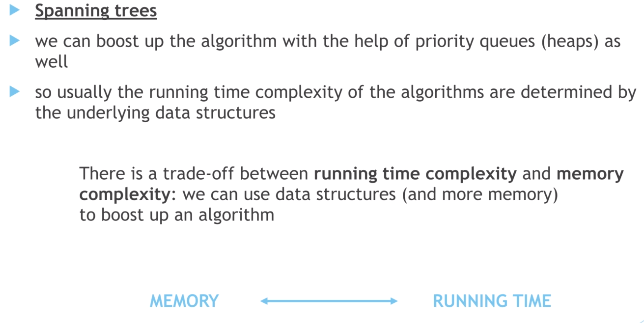
Overview:

* So we use data structures to store data in an efficient way.
* We often have the intuition that if we want to make an algorithm fast, we have to optimize it with fewer operations. We have to avoid nested for loops.
* We have to make every calculations as fast as possible. We have to use several design patterns and so on.
* But algorithms can be boosted up by proper data structures. So it is a famous quote that bad programmers vary about the code and good programmers worry about data structures and their relationships. So if we want to make sure that the given application is going to be fast enough, then we have to deal with the data structures.
* We have to find the right data structures to use so data structures make sure the running time complexity will be better.

Let's consider two examples.

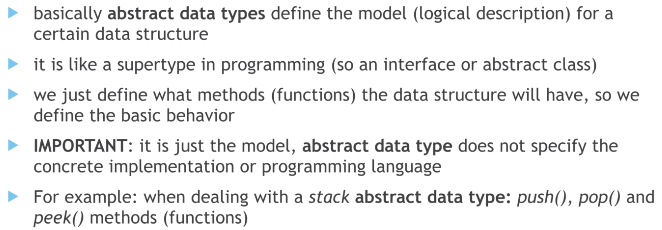


By the way, there is a tradeoff between running time, complexity and memory complexity.



What is the crucial difference between data structures and abstruct data types.

what are abstract data types?



What's crucial that it is just the model itself.

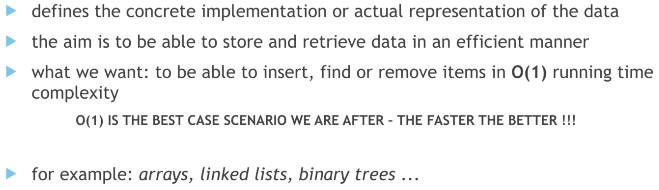
Abstract data type doesn't specify the concrete implementation or the programming language.

For example, when we are dealing with the stack obstruct data type. There are three methods or functions. Push Pop and peek as far as the abstract data type is concerned.

We are just going to define that. There must be three methods that push the Pop and the peak.

But the abstract data type is not going to define the concrete implementation. It has something to do with the behavior and what methods or functions to have as far as data structures are concerned.

**Data structures**



This is why we are going to talk about error rates than binary search trees, than hash tables or dictionaries in Python.

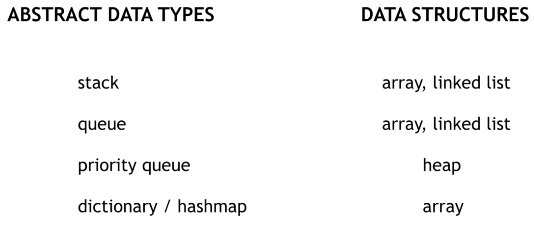
Because with the help of dictionaries. Finally we are able to achieve or the one constant running time. Complexity for these operations. if we have a perfect hash function anyways, we are going to talk a lot about these data structures.

What's crucial that the abstract data type defines the behavior and the data structures have something to do with the concrete implementation of this behavior.

So, for example, arrays, Lindley's binary search trees, AVR trees or hash maps or dictionaries are

data structures.

Summarize:



there are abstract data types that defines the behaviors or the logical descriptions of data structures, and we have the concrete data structures.

So, for example, we have the stack as an abstract data type and we will see that we are able to implement the stack with the help of an array or with the help of a linked list.

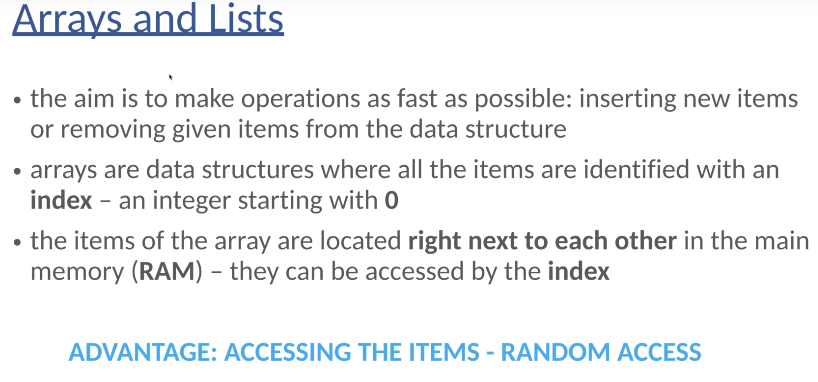
We can do the same with Queue obstruct data type. Q is the logical description and we can implement it with the help of an array or a LL.

Then we have a priority queue as the obstruct data type and heap as the concrete implementation.

So the concrete data structure dictionary in Python or Hashmap in Java. These are the abstract data types that defines that behavior without the implementation. And we can implement a dictionary or a hash map with the help of a one dimensional array. This is the concrete implementation.

\*\*\* Data structures- Arrays \*\*\*

So first of all, arrays are data structures.

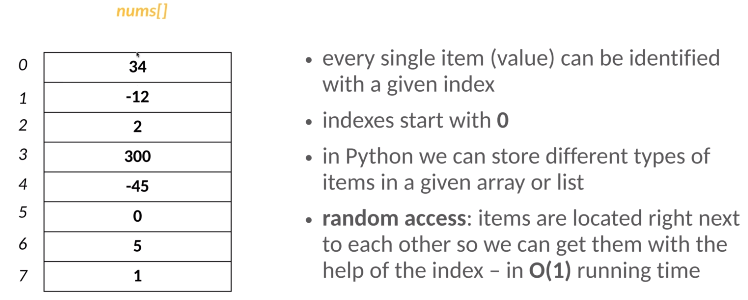


It is quite misleading in Python because lists are basically arrays. Python is implemented in the C programming language. And if you take a look at the documentation awfullest data structure, then you will come to the conclusion that lists in Python are basically dynamic arrays.

This is the main advantage of arrays.

* Every single item in the array is associated with a given index. And in this it starts with zero.
* As we have discussed in the previous lectures, the items of the array are located right next to each other in the main memory or in the RAM. This is the reason why they can be accessed with the help of an index.
* So the main advantage of arrays is the so-called random access feature that we can access items based on indexes.

Example :



In this case, we are storing integers and as you can see, every single item has a given index.

The first item has index zero. Second item has index one, the third item has index two and so on.

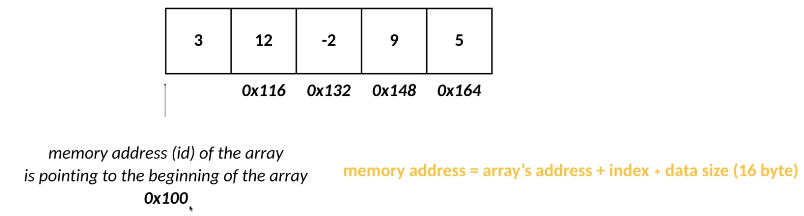
We can store different types of items in a given array or list. So it is quite common in programming languages such as C, C++ or Java that we have to define the type of items we are going to store so we can store integers or we can store strings. We can store a floating point numbers, but we are not able to store different types of items in other programming languages.

As far as Python is concerned, we can store different types of items. For example, the first item can be an integer. The second item can be a custom object. The third item can be a string. The fourth item can be a floating point number. So this is why Python as a programming language is extremely powerful and convenient.

And we can take advantage of the random access feature of arrays. So items are located right next to each other. So we can get these items with the help of their indices. This is why or the one constant running time complexity can be guaranteed.

So what does it mean that if we know the index of a given item, for example, we would like to get the item with index for that and we can get to that item based on the index. So this operation has ordered the one constant running time complexity.

OK, so what do we show that, for example, if we want to create a one dimensional array for four items, then these four items are going to be right next to each other in the so called RAM random access memory. So this is how we can allocate memory for four items. And because these items are right next to each other, we can use so-called indices, the memory address or the idea of the array is pointing to the beginning of the array.



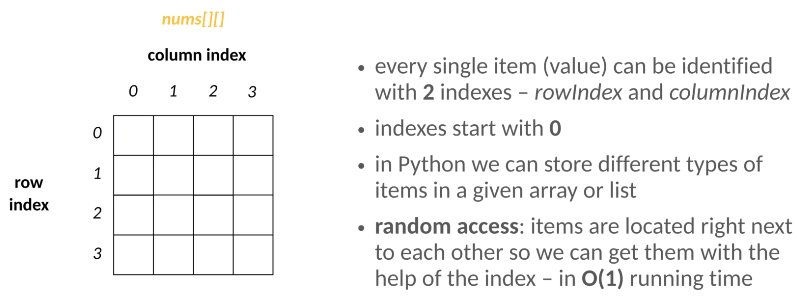
For example, the address of the array is zero one zero.

Then we have five items in the one dimensional array and because they are stored right next to each other in the memory, we can calculate the index of these items quite easily.

So this is why the memory address of a given item is equal to the array's address. So zero acts one zero plus the index of that item.

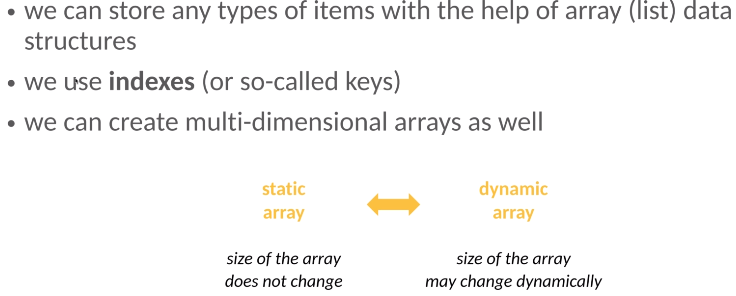
The index is zero for the first item, one for the second item, two for the third item and so on. So plus the index times the data size.

So the size of a single item, we can use multidimensional arrays.



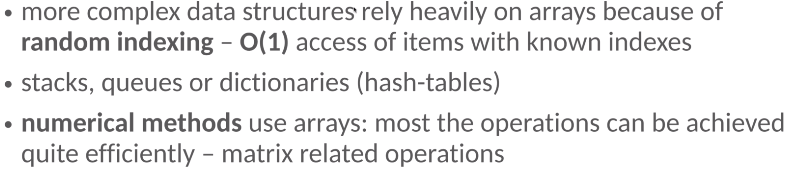
If we have a two dimensional array, then every single item can be identified with two indices, row index and the column index.

This is the main advantage of array's that if we know the index of that given item, then we can get it quite fast in order one constant running time complexity.



And what's crucial is that there are two types of arrays, static arrays, when the size of the array doesn't change or we have dynamic arrays, when the size of the array may change dynamically.

**Applications of arrays:**



More complex data structures rely heavily on arrays because of random indexing. So because of this random indexing, we can achieve or the one constant running time complexity. So this is a huge advantage of arrays.

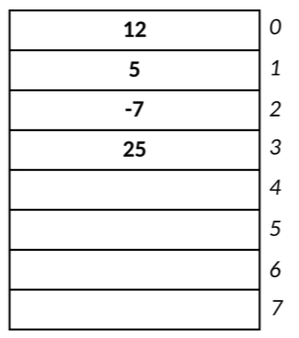
And this is why more complex data structures rely heavily on arrays such as stacks to use or dictionaries or so-called hash tables.

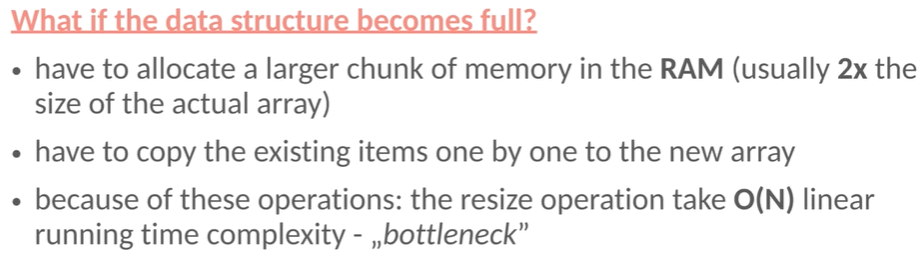
So most of the numerical methods are implemented with the help of one or multi-dimensional arrays.

Array -Operations



So what do we have to do? Basically, we just have to insert the items at the end of the array data structure.





So this is the bottleneck operation as far as arrays are concerned. So there is a huge tradeoff between the amount of memory we use and running time of the given data structure, because if we start with a small array, then we do not waste memory.



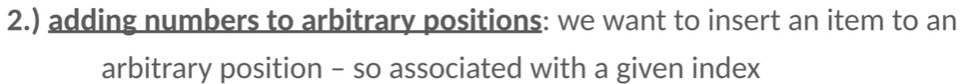
But we have to resize the array, often with O(N) linear running time complexity.

On the other hand, if we allocate a huge array at the beginning, then we do waste memory because of the large size.

But at least we don't have to bother with the resize operation. This is a typical trade off in computer science. As for memory and running, times are concerned.

If we want to make an algorithm fast, then we have to make sure that the algorithm uses more memory. If we want to minimize the amount of memory the given application uses than the running time, complexity is not going to be the best one possible, which means that the algorithm is going to be slower so we can boost up the running time of an algorithm with the help of extra memory so we can come to the conclusion that we can make a given application faster with additional memory.

what if we would like to insert a new item to an arbitrary position?

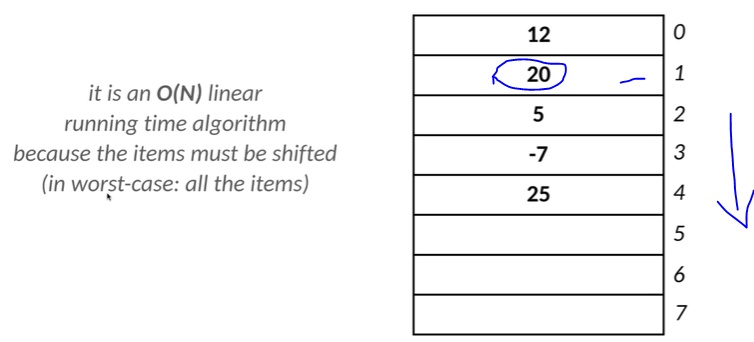


So we want to insert an item associated with a given index, for example. We would like to insert a new value with index one = 20. There's already an item.

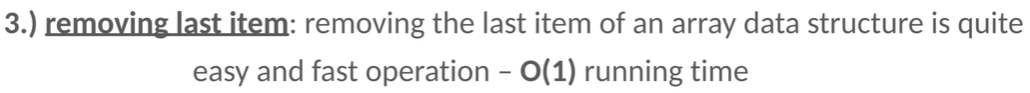
First of all, we have to shift every single item that has higher index values than the actual one.

That 25 will be shifted, minus seven will be shifted, five will be shifted, and then we are able to insert the new value. This operation has auto and linear running time because the items must be shifted.

And in the worst case scenario, if we want to insert a new item with index zero, then we have to shift all the items.

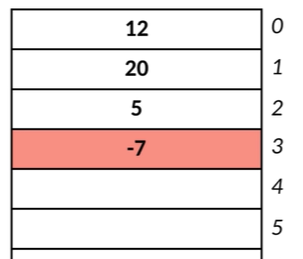


What if we would like to remove the last item from an array data structure?



We can do it in order one constant running time complexity.

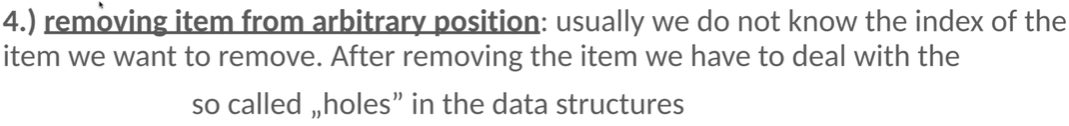
So for example, we have these values and we would like to remove the last item.



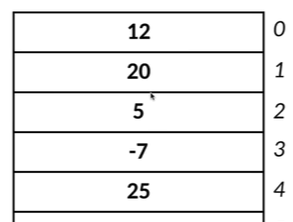
We just have to remove it again. We would like to remove the last item. We just have to remove minus seven and so on.

So removing the last item again is quite fast.

We would like to remove an item from an arbitrary position.



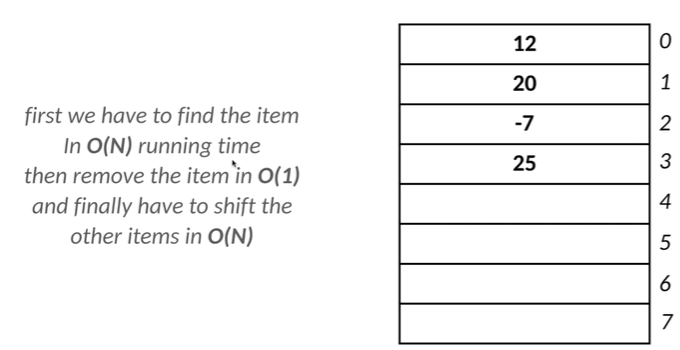
example, we would like to remove the number five.



First of all, we have to search for that given item. It is going to be in order (N) linear running time operation. Then after removing the item, it is the so-called hole in the data structure.

So as you can see, it is not favorable.

So what do we have to do? We have to make sure that the items are going to be shifted one step upwards.



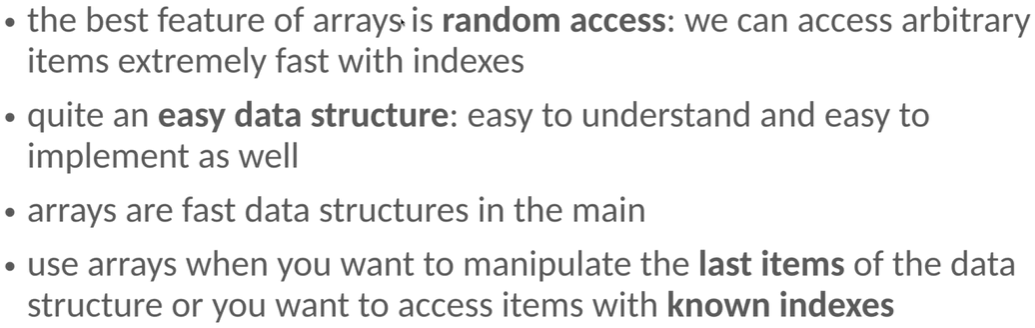
So we can come to the conclusion that if we want to manipulate the last item of an array data structure where we insert the new item or we remove a given item, then it is going to be fine.



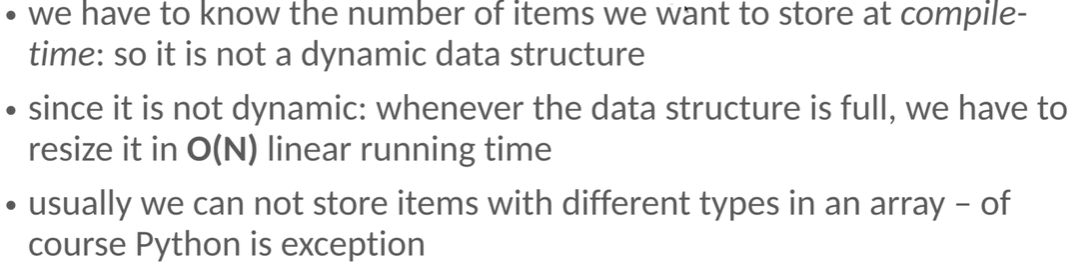
This is why we like arrays, but if we want to manipulate arbitrary items, whether we are insert.

New items at arbitrary positions or remove a given item from an arbitrary position, these operations will have O(N) and running time complexity so we can come to the conclusion that if these kinds of operations will dominate in our application, then array data structure may not be the best option possible. Because of this linear running time complexity, we can do better than linear running time.

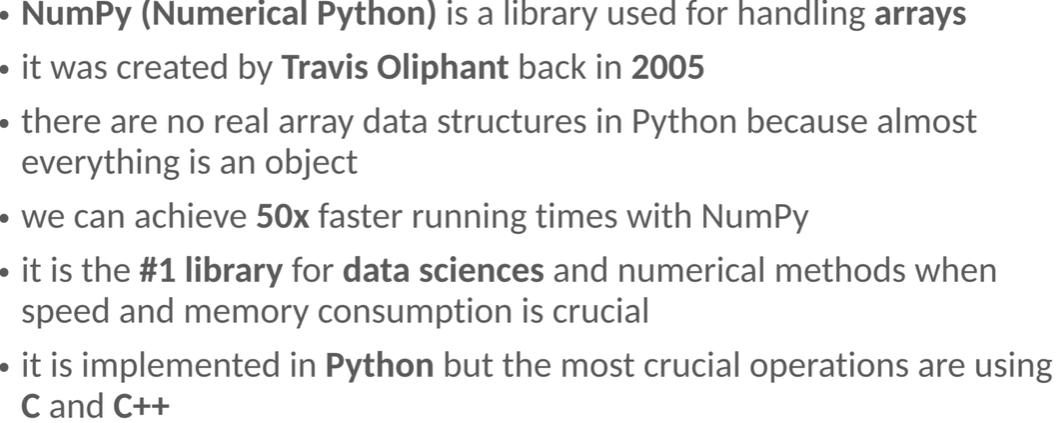
summarize the advantages of arrays.

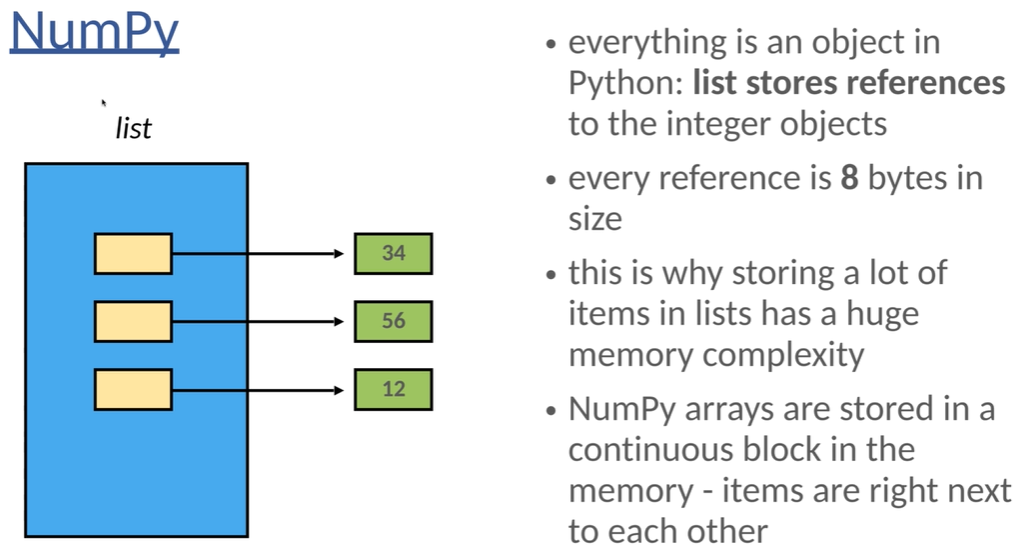


what about the disadvantages?



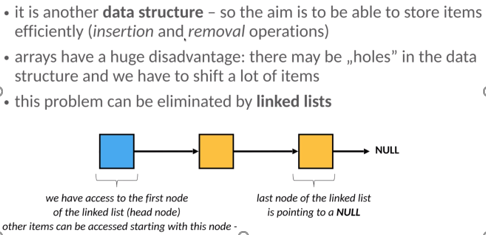
What are list in Python

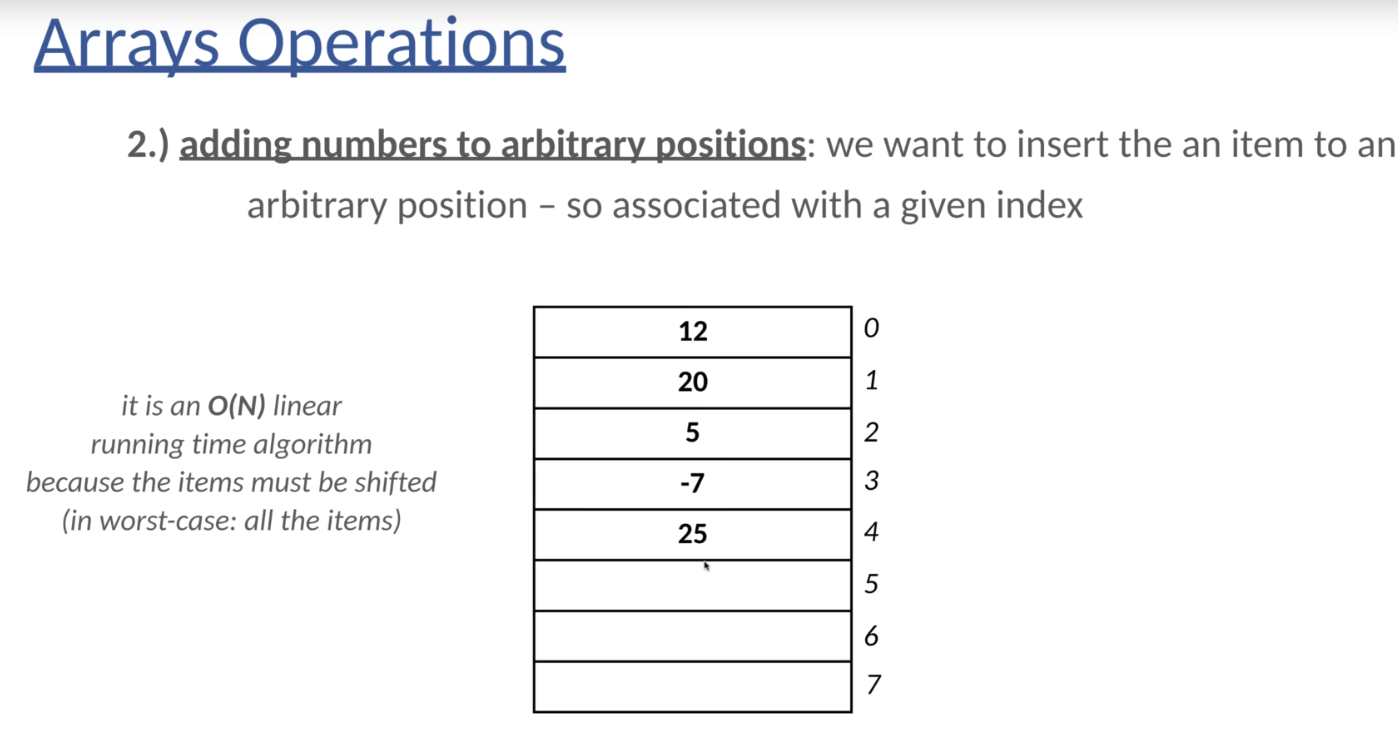




**Data Structures – Linked List**

Linked lists are data structures.





We would like to insert items as fast as possible and we would like to remove items from the list as fast as possible.

We have been talking about arrays and we came to the conclusion that arrays are working extremely fine if we know the index of the item, we would like to manipulate.

But there may be several issues. example, if we want to insert value 20 in the array data structure with index one, then what's going to happen?

Because there is already a value in the array data structure with index one. Why do we have to do we have to shift all the items one step downwards in order to be able to insert the value 20?

So it has a O(N) linear running time complexity because the items must be shifted. And in a worst case scenario, we have to shift all the items if we want to insert a new item with index zero.

And this is exactly why linked list data structures came to be, because it is extremely efficient to manipulate the first items when using LL data structures.

so arrays have a huge disadvantage, which means that there may be holes in the data structure and we have to shift several items. And this problem can be eliminated by Linked lists because in the Linked data structure we have independent nodes. These nodes are going to contain the given information, for example, integers, floating point numbers, custom objects and so on.

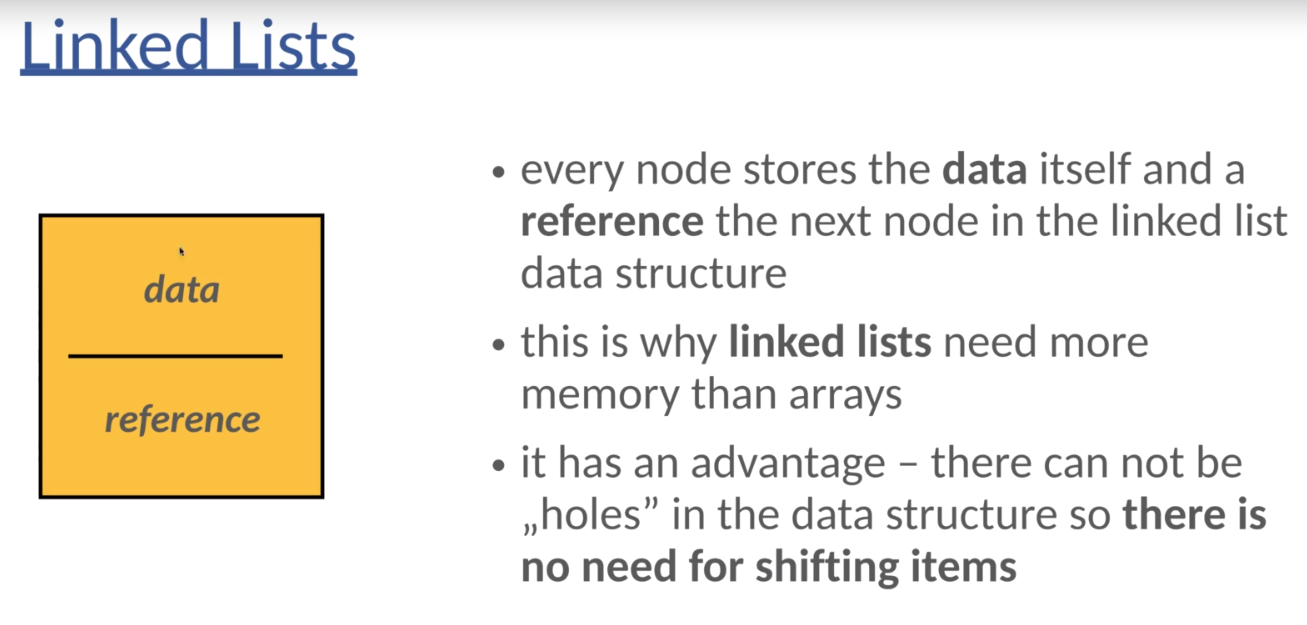
And what's extremely crucial that these nodes are connected by references.

So as far as linked list data structure is concerned, we have access to the first node of the list exclusively. This is called the head node, and other items can be accessed starting with this node. And what's crucial that the last node of the list is pointing to NULL .

So basically, this is how we can traverse a link list that we have to start with the head node.

And then we have to follow the references until we come to the conclusion that the reference is pointing to a null.

This is when we know for certain that it is the last node in the link data structure.

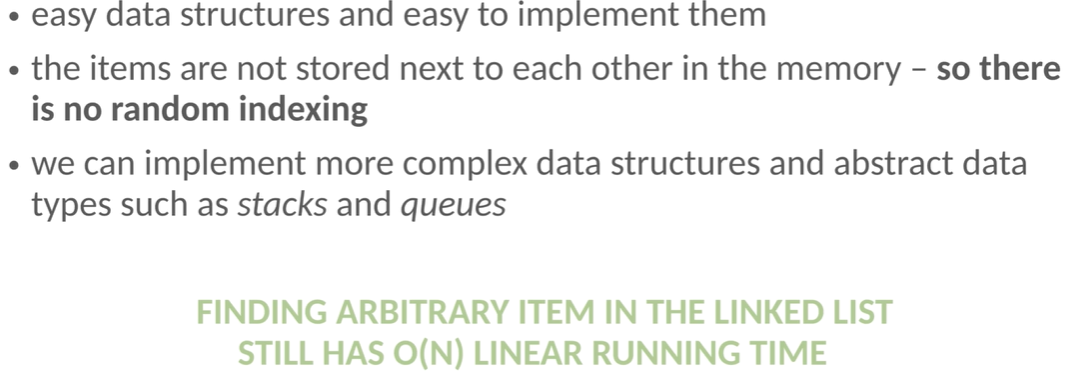


single node stores a data, each can be an integer, it can be multiple floating point numbers, it can be a Boolean variable, it can be a custom object, and it's towards a given reference pointing to another node in the link list.

So every nodes towards the data itself and the reference to the next node in the link is data structure. And this is why linked list needs more memory than arrays because we have to store the pointers pointing to the next node in the data structure.

But it has a huge advantage, which means that there cannot be holes in the data structure. So there is no need for shifting the items.

And this is why O(N) linear running time complexity approaches can be reduced to O(1) constant running time complexity.



It is quite easy to understand what's happening under the hood. Basically, we have nodes in a long list storing the given data and every single node has a so-called reference that is pointing to a next node in the data structure. And that's all about the theory behind linked list.

So it is quite easy to understand and it is quite easy to implement.

It was crucial that the items are not stored right next to each other in the main memory. So there is no random indexing.

So we are not able to access the given items based on a given indices in order one constant running time complexity.

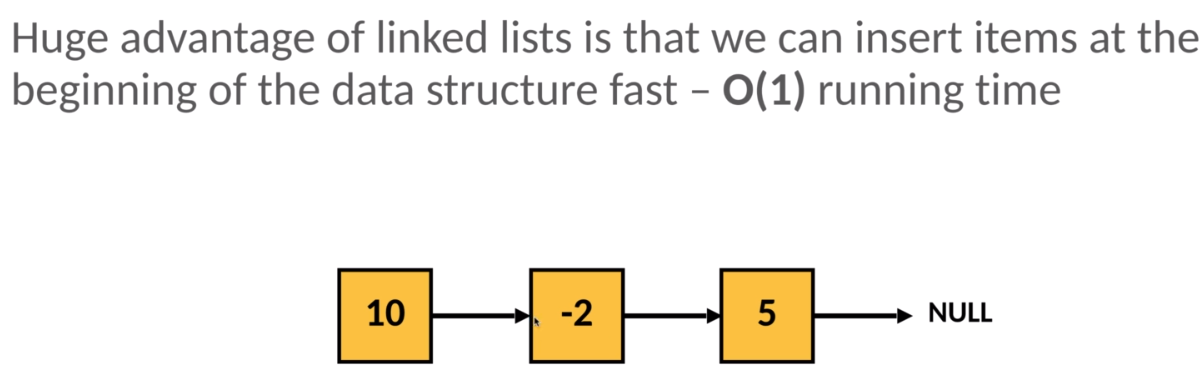
And this is one of the main disadvantages of linguist's. So as far as arrays are concerned, we have a random indexing.

But it is not true for Linked List because the items are not stored right next to each other in the memory, and we can implement more complex data structures and abstract data types with Linked list such as Stacks and Qs.

The main problem is that finding an arbitrary item in a link list still has O(N) linear running time complexity.

And this is exactly why we have to talk about more complex data structures such as binary search trees and hash tables or dictionaries in Python.

Linked List Operations:  
A huge advantage of LL is that we can insert items at the beginning of the data structure quite fast in O(1) running time complexity, because we just have to update the references.



Ex: if we want to insert 5, then this is going to be the first item. It is pointing to NULL at the beginning. Then what if we would like to insert minus two, then -2 is going to be the first node in the data structure and it is going to point to number five. what if we would like to insert 10, then 10 is going to be the first item of the data structure and we just have to handle the references. It is pointing to the previous first item of the data structure, minus two. Then, for example, we can insert value to you and after the insert operation, it is going to be the first item of the LL data structure.

Why O(1)? Because this is exactly how we designed the structure of LL.

we store a reference to the first node in the data structure. This is the so-called head node.

So every other item can be accessed starting with the first node.

if we want to insert new items at the beginning of the LL data structure, then we have to manipulate the first node. So basically the head node and because we have access to that given node, we can update the pointers in order one constant running time complexity.

So this is by manipulating the first item of the list is extremely fast.

What if we would like to remove a given item from the beginning of the list?

Again, because we have a reference to the first node, the so-called head node of the list. Again, it is going to be quite a fast operation. It can be done in O(1) constant running time complexity.

So basically because we have an active reference to the first node of the data structure and we can update the first node of the data structure quite fast. This is exactly why we stored a reference to the head node. there are slow operations as well as far as LL are concerned.

For example, if we want to insert a new item at the end of the list or basically if we want to insert a new item to an arbitrary location, what do we have to do?

We have to start with the head node. So the first node of the link list, because this is the node we can access, we can access the head node exclusively.

We are not able to access the center node. We are not able to access the last node.

This is the topology of the data structure. We can access the first node exclusively so we have no other option. But to start with the beginning. It is not the last node because we know for certain that the last node is pointing to NULL.

So we keep going. Ten is the last item. not. We keep going. Minus two is the last item. not. We keep going. Five is the last item. So we have to insert 19 to be the last item of the data structure.

And if we want to insert the new nodes so we know the location, then it is a fast operation. We just have to update the references. Five is going to be pointing to 19 and 19 will be pointing to NUll.

What's extremely crucial that the slow operation is to find the given node.

So we have to find the item we want to remove or we want to insert with a linear search in order(N) linear running time complexity. And it is absolutely crucial that we can access the head node exclusively.

So we always have to start with the first node of the data structure. And it is the same if we want to remove an arbitrary item.

2 -> 10->2 -5 -> 19

For example, if we want to remove the number five, then what do we have to do?

We have to start with the first node and we have to keep looking for that item. So we have to consider the next node in the LL data structure until we find. The item we are looking for and then we can remove it again. We just have to update the references, it is fast. What makes it slow is that first we have to traverse the long list on a one by one basis.

So this is why LL are working extremely fast if we want to update or manipulate the first item.

But if we want to update or manipulate the last item or an arbitrary item, then the running time complexity is going to be linear, which is not that good. We can do better. **binary search trees.**

OK, so we can come to the conclusion that if we want to manipulate the first item, whether we want to insert or remove the first item, it is extremely fast. It has O(1) constant running time complexity. This is exactly why we like LL.

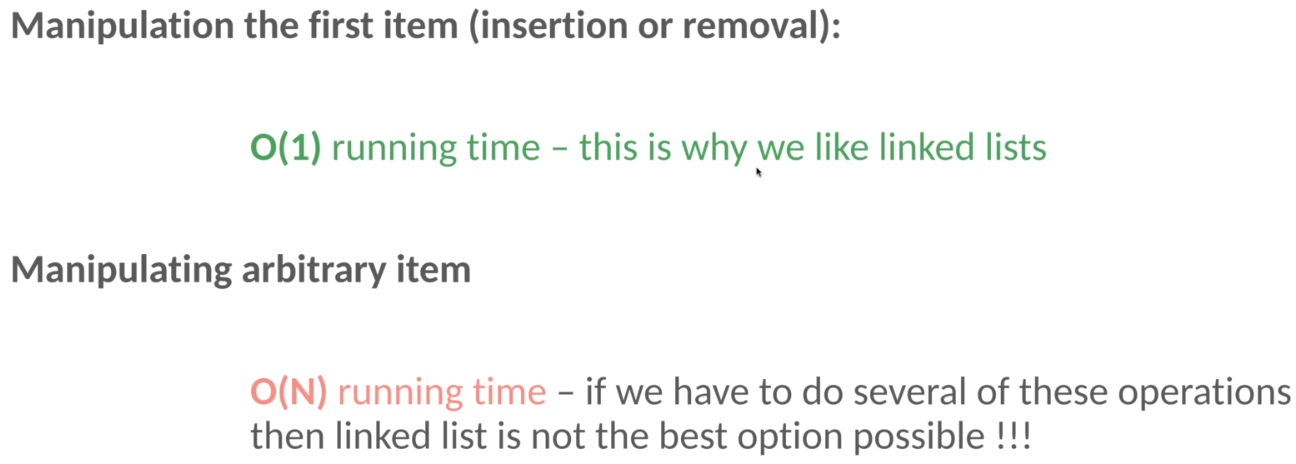
But what if we would like to manipulate arbitrary items?

For example, the last item of the data structure where we would like to insert a new item or we would like to remove an existing item, it's going to have o(n) linear running type complexity, which means that if we have to do several of these operations, then LL is not the best option possible.

**Advantages and disadvantages of LL.**

* LL are dynamic data structures.
* They can acquire a memory at runtime by inserting new nodes.

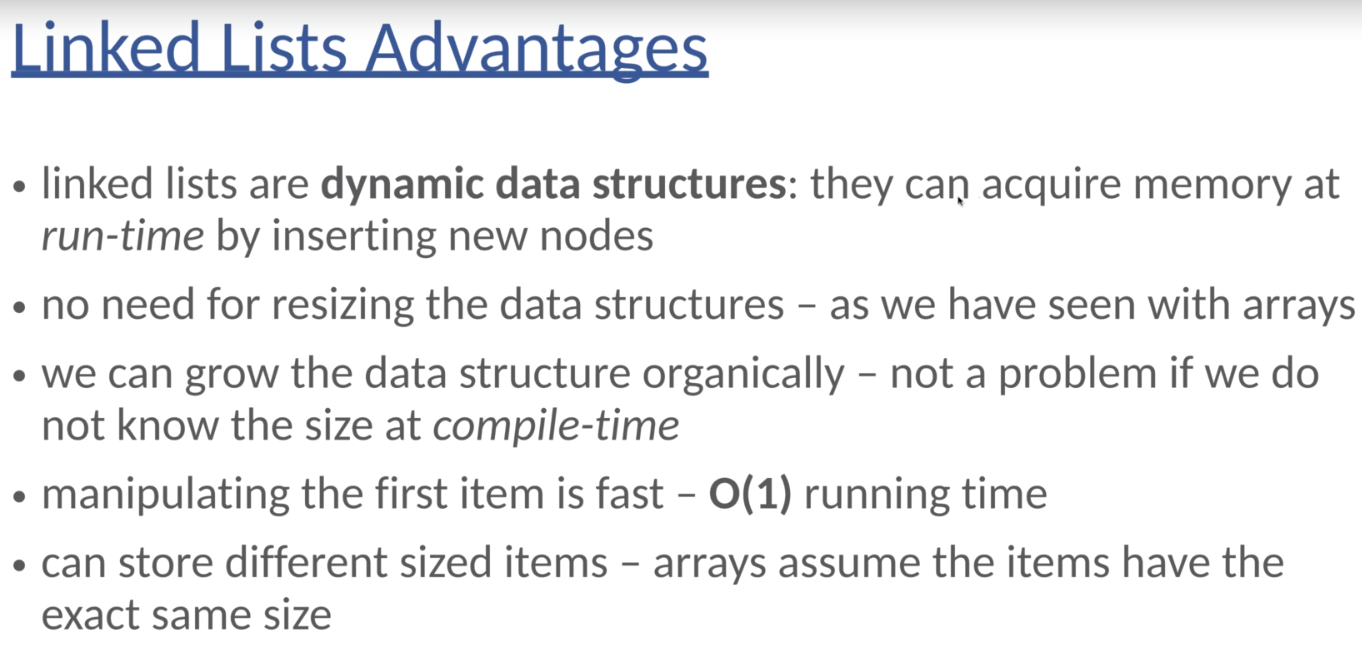
So what does it mean? : There is no need for resizing the data structure, as we have seen with arrays. And it is absolutely crucial that this data structure can grow organically.

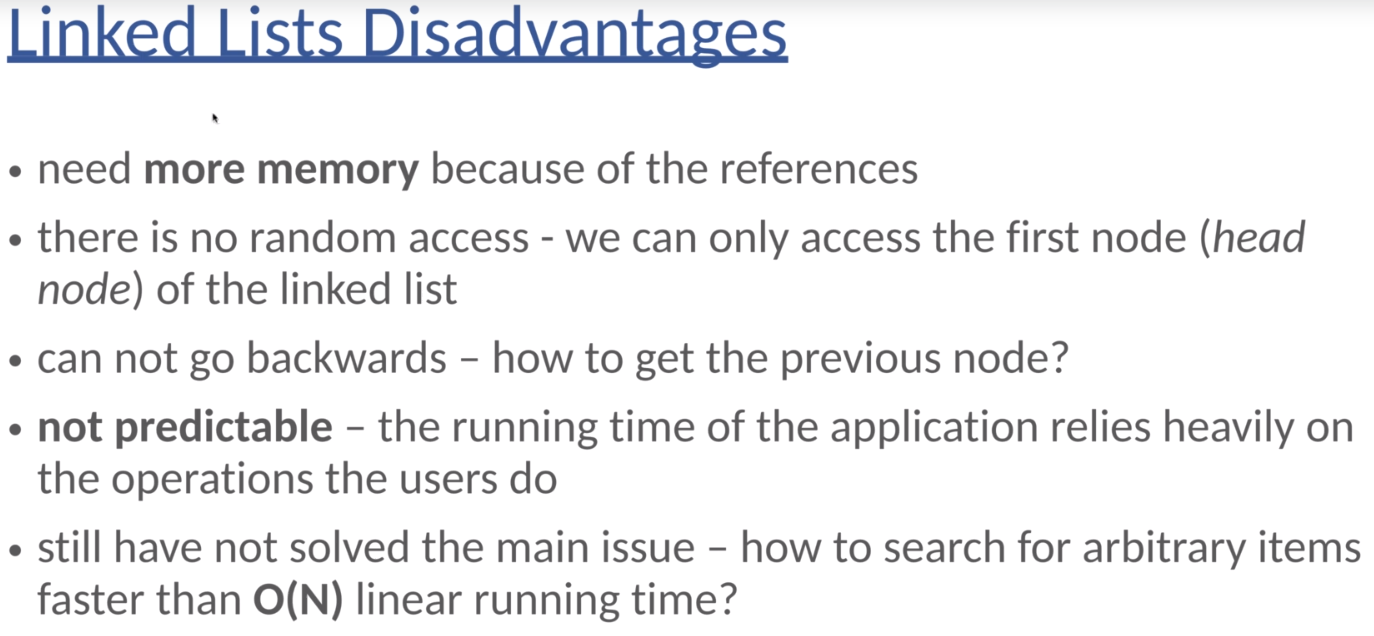


It is absolutely not a problem if we do not know the size of the LL at compile time, we are able to insert new nodes without any problem in O(1) Constant running from complexity.

It is not the same as we have seen with arrays. As far as arrays is concerned, we should know the number of items we will have to store because otherwise we have to resize the area, which is quite an expensive operation with O(N) and linear running time complexity.

Manipulating the first item of a LL is extremely fast and wants to show that we can store items with different sizes. Arrays assume that items have the exact same size and it is not true with LL. So LL are dynamic data structures. Why arrays are static data structures.





**Revisiting remove operation**

In the theoretical section we stated that remove operation is quite fast with linked lists. But during the implementation we saw that it has **O(N)** linear running time complexity - which is not that fast. Why?

The answer is that usually when we use linked lists - we assume that we have to remove (and insert) the first item which is the head node that can be accessed in **O(1)** constant running time.

* the remove function we have implemented is totally fine
* if we use it to remove arbitrary item then first we have to find it and then we have to update the references (exactly how we have implemented it) - with **O(N)** running time
* if we want to remove the first node (head node) with the same function - then there is no need for the while loop (Python will exit the while loop after the first iteration) so the final running time is **O(1)**

**CONCLUSION: WE CAN REMOVE THE FIRST NODE OF THE LINKED LIST EXTREMELY FAST - WITH O(1) CONSTANT RUNNING TIME**

This is exactly the reason why we have linked list data structure - to be able to manipulate the first item extremely fast.

Comparing Linked List and Array:

1. Dynamic and static data structures.

* arrays are static data structures. - Usually we have to know the size of the data structures in advance. if we do not know the number of items we have to store, then we can resize the given underlying data structure.

So it is not a big issue. But anyways, usually arrays are considered to be static data structures because if the number of items we have to store change, then we have to change the data structure itself.

By the way, we are going to talk about hash maps or dictionaries and we will come to the conclusion that we have to resize the underlying one dimensional array data structure. So this is why arrays are considered to be static data structures.

* Linked list are dynamic data structures.- We don't have to resize the given data structure, which means that they can grow organically based on the references. There is no resize operation at all because we just have two of the references or the pointers. So LL are way better because they are dynamic data structures. There is no need for the resize operation.

2. what about random access(Random Indexing)?

* Random access is a huge advantage of arrays. arrays are concerned, the items are located right next to each other in the main memory. And this is why we can use indices in order to identify and find the given items in O(1), constant running time complexity.
* Unfortunately, we are not able to use random access with linked lists because the items are located in different parts of the memory. So there is no random access in LL data structures. So as far as random access is concerned, arrays are way better than Linked lists.

3. manipulating the first items of the data structure?

* we have to shift several items or all the items in a worst case scenario when manipulating the first items in an array data structure. This is why if we manipulate the first items of an array data structure, then the running time complexity will be linear.
* We can do better with lists because LL are dynamic data structures and we just have to update the references around the head.

we have access to the first item of the data structure, which is called the head node.

So as far as manipulating the first items of the data structure, link lists are better.

4. manipulating the last items?

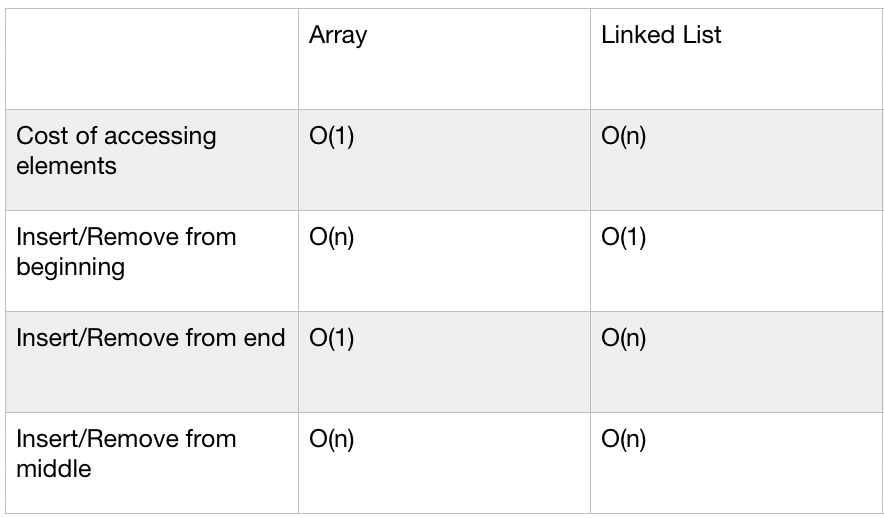
* If we want to manipulate the last items, then we should use arrays because there cannot be holes in the data structure when manipulating the last items. We just have to append that item to the array or do we have to remove the last item and it can be done in O(1) constant running time complexity, which is quite fast.
* LL have access to the first node or the head node exclusively? So in this case, if we want to manipulate the last item, what do we have to do? first of all, we have to traverse the whole list in order to get to the end of the data structure. And it can be done in O(N) and linear running time complexity. So, arrays are faster when manipulating the last items.

5. memory management?

* Arrays do not need any additional memory.
* And LL, on the other hand, do need extra memory because of the references and the pointers. So every single node in the LL data structure store a reference to the next node in the data structure. So basically LL need extra memory.

Huge disadvantage both for arrays and LL.

And it is that if we want to search for an arbitrary item when we do not know the index of that item or we want to remove an arbitrary item, then it takes O(N) linear running time complexity for both data structures. So this is the main disadvantage for both data structures. And this is why we have to talk about more complex data structures such as binary search trees.



Linked List Application  
so usually operating systems use almost all the important data structures from LL to hash maps.

1. So, first of all, as far as low level memory management is concerned, linguists prove to be very, very important. For example, when dealing with C programming related malloc() and free functions, I'm not sure whether you are familiar with C programming, but unlike Java and Python, we have to deal with memory management.

So with the help of the malloc() function, we are able to allocate memory on the heap and with the help of the free function, we are able to free the memory on the heap.

So it is built in functions. We can manipulate the heap memory.

For example, if we want to allocate thirty bytes of memory in the heap, we just have to use this code snippet.

Char \* char\_ptr = (char \*) malloc(30) //allocate 30 bytes of memory.

So if we use pointers, we are going to manipulate the heap memory.

And these data blocks are linked together with the help of double-dealing lists.

So the header of every single block contain information related to that given block and stores the reference to the next block in the heap memory and vice versa.

So the next block contains a reference to the previous block and to the next block. So that's why it is going to be a double dealing list if you are interested in the details.

So this is why it's crucial that link list of free blocks will be traversed, which means that the heap memory is represented as a double dealing list, basically.

2. applications of Windows.

If you use all tab, then you can take a look at the active tabs on your machine.

In this case, I had several tabs, such as the PowerPoint slides such as this heap management such as notepad plus such as Chrome data structure in Java and so on.

So if you click on ADD tab that you can change these tabs, whatever you want, and basically these tabs are linked together with the following list.

3. L lists are used when dealing with photo viewer application as far as next photo is concerned and a previous photo is concerned.

So basically this photo viewer uses illegalize data structure under the hood and the images are linked together with the help of this link list.

If you click on the next image, then the next item will be presented in the link list. And if you click on the previous button, then the previous images will be shown. So again, this application is implemented with the help of a LL.

4. And finally, the most important application of LL data structure has something to do with block chain and the Bitcoin ecosystem.

So the block chain is a simple link list where blocks are cryptographically linked together. So as far as Bitcoin is concerned, these blocks contain the given transactions and every single block contains a given hash.

This is the identifier of the given block and every single block contains a given hash pointing to the previous block in the block chain.

So as you can see, a block chain is no more than a simple legalist where blocks are cryptographically linked together. So every node in the block chain has two hash values, the own hash and hash tag. The previous block, and this is why block chain is no more than a simple English cryptographically living together.

**\*\*\* Doubly Linked List \*\*\***

In linked lists we came to the conclusion that we have to store a reference to the first node of the data structure. So this is called the head node.

2 🡪 10 🡪 -2 🡪 5 🡪 19 New Item🡪 NULL

And there is a huge problem because, for example, if we have a long list like this, we want to insert a new item, then we have no other option but to start with the first node and we have defined at the end of the list and then we are able to insert the new item.

So this is why manipulating the last items of a simple list takes O(N) linear running time complexity, because first we have to consider all the items one by one in order to find the last node in the LL list. And basically this is exactly why Double-dealing list came to be.

So again, it is a data structure. So the aim is to be able to store items efficiently and to make sure that insertion and removal operations are going to be as fast as possible.

So in this case, we store a reference to the head to node. So the first node of the data structure, this is what we have seen when dealing with LL. But it is absolutely crucial that we store a reference to the last node of the LL list. This is the so-called tail node.

And what is absolutely crucial that every single node has two references.

So given a node is pointing to the next node and there is another reference to the previous node. So this is why the previous node of the head node is a null.

And the next node over the tail node is another pointer again. And otherwise every single node keeps a reference to the previous node and keeps a reference to the next node in the link. This is why it is called Double-dealing List.

Every single node in a dealing list has a data. It can be an integer, a floating point number, a string, a custom object. So the same as we have seen a standard link lists and every single note has a reference to the next node and to the previous node.

So this is why it is called Double-dealing List, because now we have references to the next node and to the previous node in the LL list.

And by the way, this is why Double-dealing lists need even more memory than standard linked lists. But there are several advantages.

For example, there is no need for shifting items because we store a reference to the first item and to the last item so we can manipulate the first items and the last items of the data structures quite fast in order one constant running time complexity.

3🡨>2🡨>1

if we have a **Doubly Linked** list like this, then they had to know this pointing to the first node, the tail node. So the last node is pointing to the last node over the data structure. And if we want to insert, for example, of one, we keep inserting it to the end of the data structure. So usually it is the standard double-dealing list implementation that whenever we insert a new item, then it is going to be the last item of the data structure.

But because we keep a reference to the last item, basically we just have to update the references. And this is why manipulating the last item takes or the one constant running time complexity.

OK, so this is how we can insert items and we can remove items quite fast from the beginning of the list and from the end of the list.

So this is by manipulating the first and last items of a **Doubly Linked** list takes or the one constant running time complexity.

We can do it extremely fast because, we have access to the first node. This is the head node and we have access to the last note of the data structure, the so-called tail node.

Advantages of **Doubly Linked** lists,

* because, Double-Linked list is a form of LL .This is why Double-Linked list has the same advantages as standard LL. But we have some additional advantages. So because we store references to the head node and to the tail node as well, these nodes can be accessed in order one running time complexity, which means that we can manipulate the first item as well as the last item quite fast.
* What's extremely crucial, and it is a huge advantage of double dealing this, that we can traverse LL in both directions from left to right and from right to left, because, every single note has a reference to the previous node as well as to the next node. This is why we can start at the tail node and we keep visiting the previous node and the previous node and the previous node until we heaped the head.

Traverse the Double-dealing list in a backward manner, and we can start with the head and keep visiting the next node until we hit the tail node. This is how we can traverse the double-dealing list in a forward manner.

So it is a huge advantage of **Doubly Linked** lists that we can traverse the **Doubly Linked** list in both directions.

* And removing a given node is easier because there is a pointer to the previous node as well. So because we have pointers to the previous node and to the next node, this is why updating the references is a bit more easier.

But because we store more references, we need more memory. So double-dealing list is more memory heavy than simply lists. And it is a bit more complicated to implement this because we have to deal with both the pointers as far as the given pointers are concerned, pointer to the previous node and pointer to the next node.

And we still haven't solved the main issue how to search for an arbitrary item faster than linear running time complexity.

we can access the first nodes in O(1), constant running time complexity.

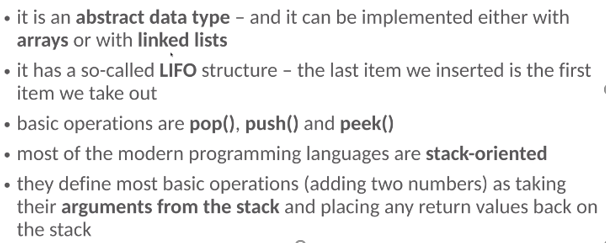
We can access the last node in order, the one constant running time complexity because we store references to these nodes.

**But what about arbitrary internal nodes?**

If we are looking for a given the value, we have no other option but to start either with the node or the tail node and consider all the items until we find the one we are looking for. Unfortunately, it has O(N) and linear running time complexity in the worst case, so we still haven't solved the main issue. binary search trees.

**\*\*\* Data Structures – Stacks \*\*\***

Stacks and queues are abstract data types. The underlying data structure can be an array or a linked list



stacks are abstract data types, which means that Stack is going to define the behavior, but it has nothing to do with the underlying implementation. So it is an abstract data type, basically a super type in programming, and it can be implemented with the help of several data structures, with the help of arrays or with the help of LL.

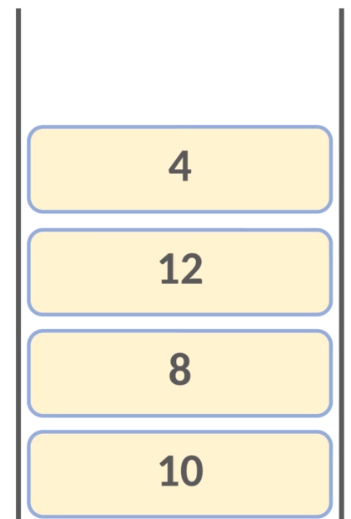
It has a so-called LIFO structure: Last item we insert is the first item we take out.

So last in, first out. there are several operations such as the Pop, the push, and the Peek operations.

With the help of the Pop operation, we can remove the last item we have inserted with the help of the push operation. We can push a new item onto the stack and with the help of the Peek, we can get the value of the last item without removing it.

So this is the crucial difference between Pop and Peek that Pop is going to remove and return with the last item. Peek is going to return with the value of the last item without removing it.

And stacks are maybe the most important abstract data types because most of the modern programming languages are STACK oriented.



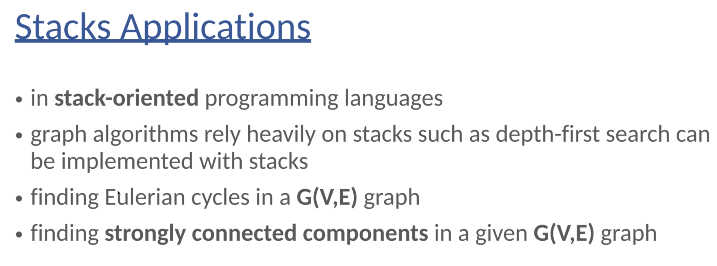
Then if we push the next item 8, then we have two items, 10 and eight. But because it is a stack, we have access to the last item we have inserted. So we are not able to remove ten, for example, because there is the eight on the top of 10.So this is why we can remove the last item we have inserted exclusively. If we insert the next item twelve, then in this case we kind of get the value of ten. But we are not able to access eight and ten directly because of LIFO structure. So the last item we have inserted is the first one we have to take out.

What about the Pop operation?

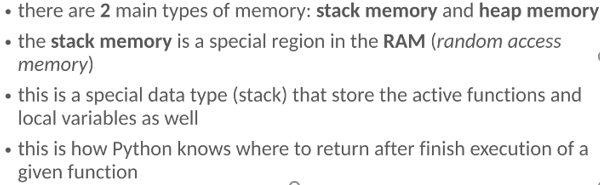
With the help of the Pop operation, we can remove the last item we have inserted.

Ex: in this case, if we call the Pop function, then it is going to return the last item we have inserted. Then again, the last item of the stack abstract data type. Then if we pop up again, then eight is going to be pulled out of the stack and finally we put the value ten. Okay, so this is how stacks work under the hood.

It is absolutely crucial that we have access to the last item of the data structure exclusively.



Stacks in Memory management



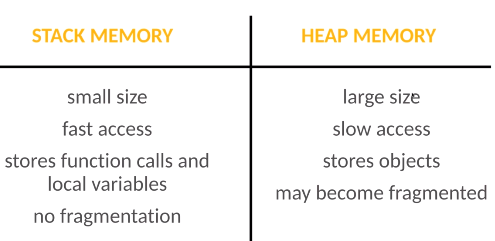
So in the random access memory, this is a special data type, a so-called stack abstract data type that stored the active functions and local variables as well.

So Python is going to store function calls and local variables in the stack memory.

what about heap memory?

Heap memory again is a special region in the RAM. The size of the heap memory is way larger than that of the stack memory. So we can store more items on the heap memory than on the stack memory. And what's crucial that we can store objects on the heap memory.

So if we want to compare stack of memory with heap memory



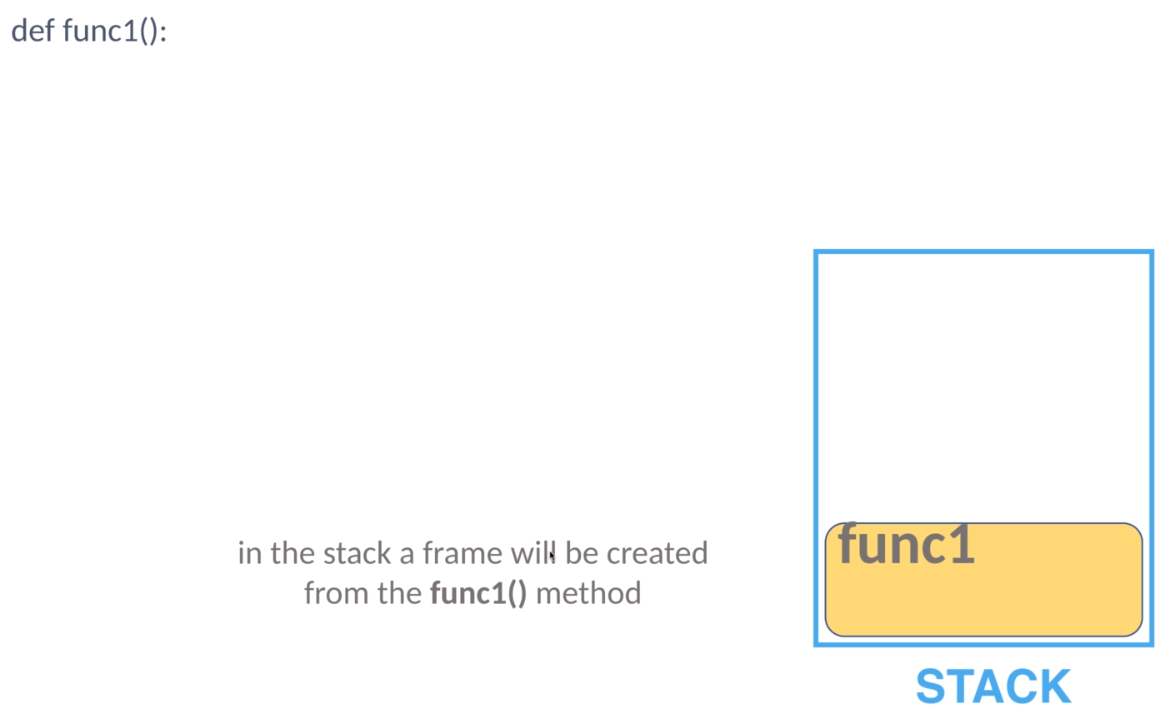
Okay, so this is the crucial difference between stack memory and heap memory.

And it was crucial that stack of memory is implemented with the help of an abstract data type, the so-called stack, the stack has a so-called LIFO structure. So the last item we have inserted is going to be the first we take out. So last in, first out structure. So Stack has something to do with an underlying data type or data structure.

A heap has nothing to do with the heap data structure. So I'm not sure whether you are familiar with data structures, but there is a so-called heap data structure. As far as the heap memory is concerned, it has nothing to do with the heap data structure. So heap memory is not implemented with a heap tree like data structure.

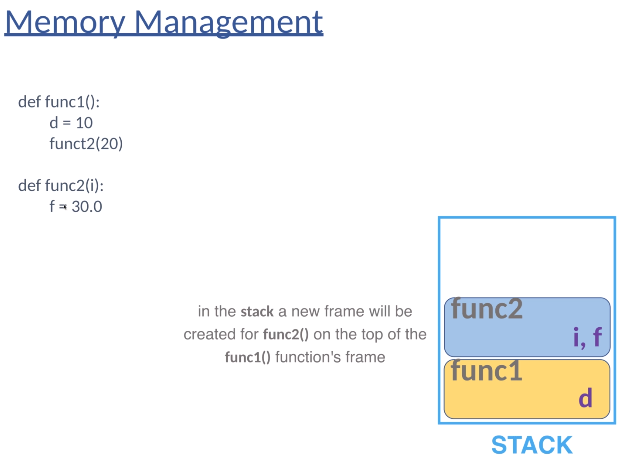
It is just a name of that type of memory.

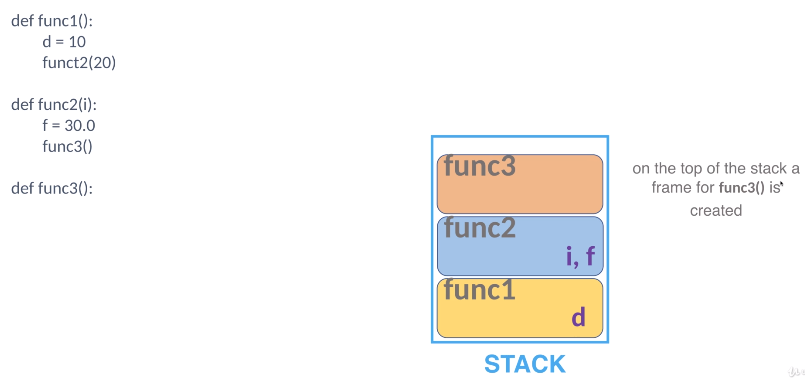
Stack Memory Visualization



So, for example, if we have a given function, then Python has to track what function has been called and these function calls are stored on the stack of memory.

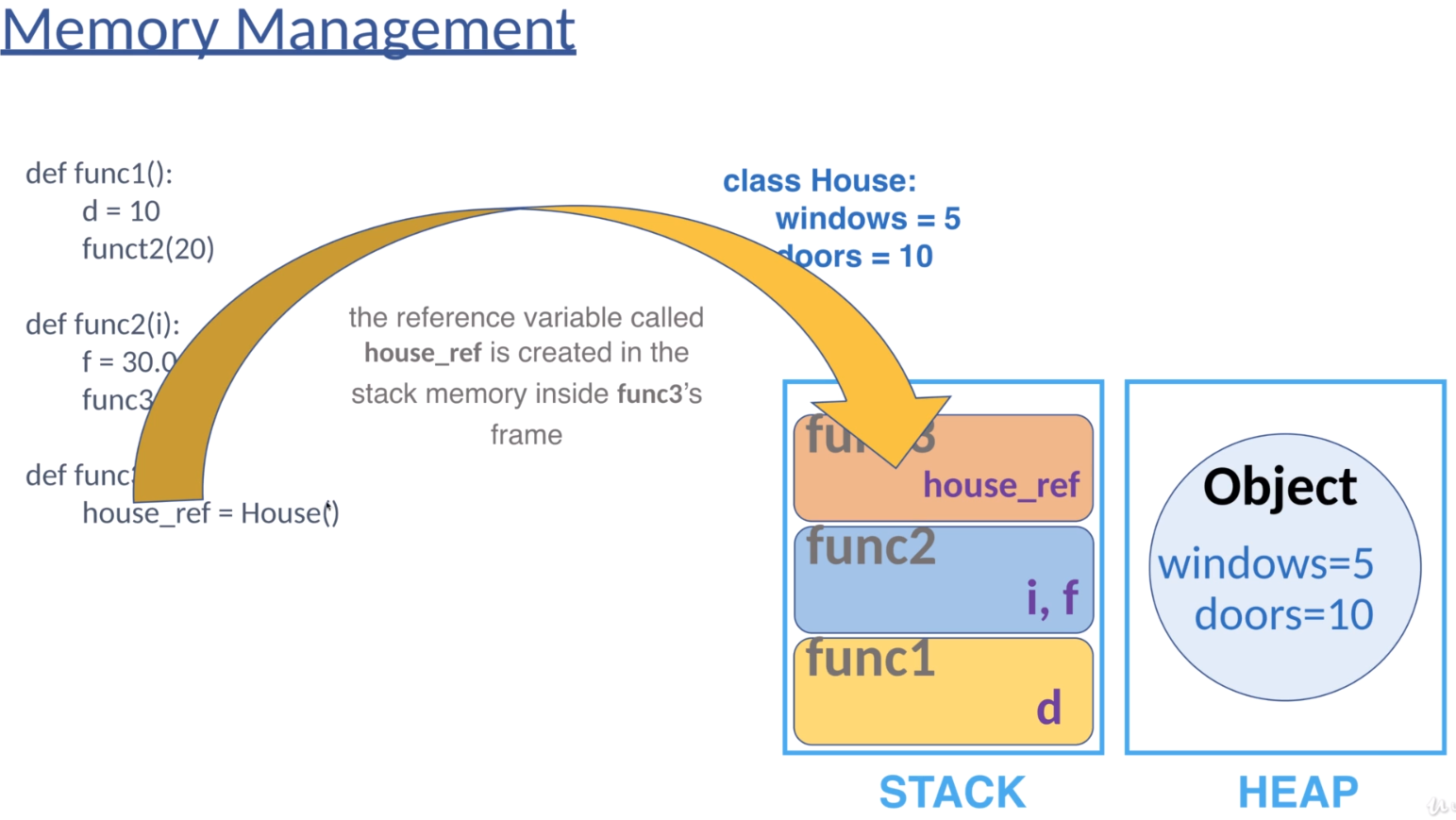
This is exactly what we have been talking about that actually functions and local variables are stored on the stack of memory.





This is how Python knows where to return after finished execution of a given function.

So in the stack, a frame will be created from the function of one method. we have a local variable D equals to 10. These local variables are stored on the stack of memory. So a local variable D in function one will also be created in the function of one frame in the stack of memory.



But what if we have a class, for example, a class house with class variables, number of windows and number of doors?

These variables have initial values, five and ten accordingly. So if we instantiate a house, then this is going to be the so-called object or the reference this house variable.

In Heap memory we came to the conclusion that objects are stored on the heap memory, which means that if we instantiate a house class, then it is going to be an object and objects are stored on the heap memory with all the class related variables, windows, and doors.

So as you can see, we have the stack memory with the function cause parameters and local variables, and we have the heap memory for objects and the object related class variable such as windows or doors. By the way, no matter that we are defining class variables or instance variables, these variables are stored on the heap memory associated with the given object. And the reference is going to be stored on the stack memory associated with the search function.

So it is extremely crucial that here we have a given object. We have instantiated with the class variable. It's five and ten accordingly.

And we have a reference, the so-called House reference that is pointing to the object on the heap memory. So that variable or the reference is stored on the stack memory associated with the given function in this case, function three. And this reference is pointing to a memory location on the heap memory, which is the instantiated house

object with the class variable as a windows and doors. OK, after instantiating a new house object, the function three execution is completed, so Python is going to return to the function T.

So this is how a programming language works.These commands are executed on a line by line basis.

Python is going to create the variable, then it is going to cause function to python that will create variable F and it is going to call function three.

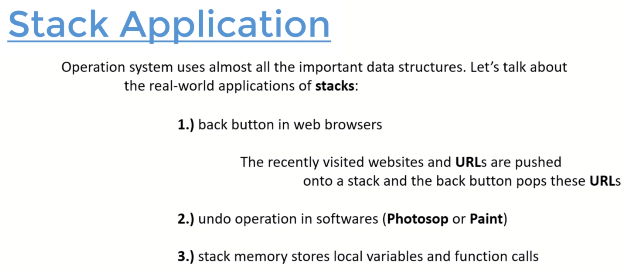
Then Python will instantiate a new house class and finally it is going to return to function to. So when functions re execution is completed, the flow of the control will go back to function two. And it was going to happen. Python is going to remove the functions three from the stack memory because the execution is completed.

Functions three is going to be removed with the reference.

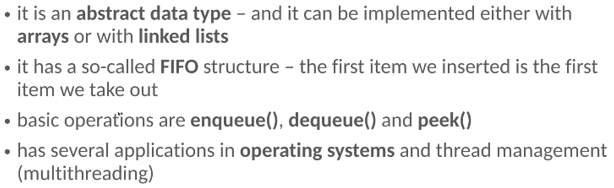
What does it mean that there's no reference on the stack of memory pointing to the object on the heap memory? So since the house reference value. It is no longer pointing to the object in the heap memory, it can be removed by garbage collection.

What's crucial, that garbage collection is going to remove unused objects from the heap memory. What does it mean that unused objects, when an object doesn't have an active reference from the stack memory that gives an object is eligible for garbage collection and Python is going to remove it from the heap memory?

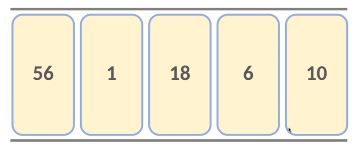
So function too as a frame on the stack memory is going to be removed with the local variables. I and F OK, so we have returned to function one. But as you can see, there are no more operations. So it means that Python is going to finish the execution of function one. So the frame is going to be removed from the stack memory.



**\*\*\* Data Structures – Queues \*\*\***



So the queue data structure is going to process the items in a FIFO manner.

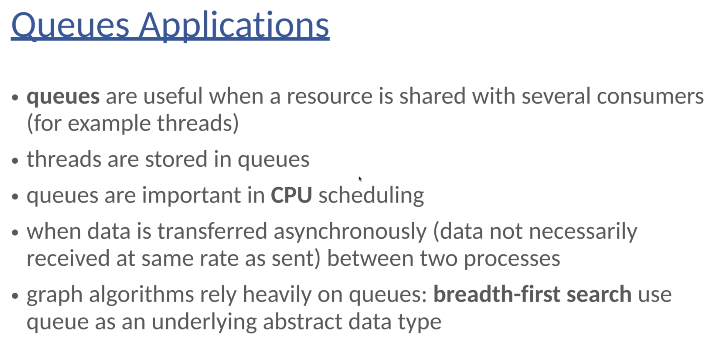


So if we insert the first item ten, then we insert the second item six, then we have 18, then we have one and finally we insert fifty six. we keep putting the items from left to right in the queue abstract data type. it is going to define the order of the items.

But whenever we remove the items we take the first item we have inserted.

So we are not going to remove 18 or one or 56 in the first iteration or we remove ten, then we can remove the first item and we have inserted, which is six, then 18, one and 56.

So if this is how the Q abstract data types work.



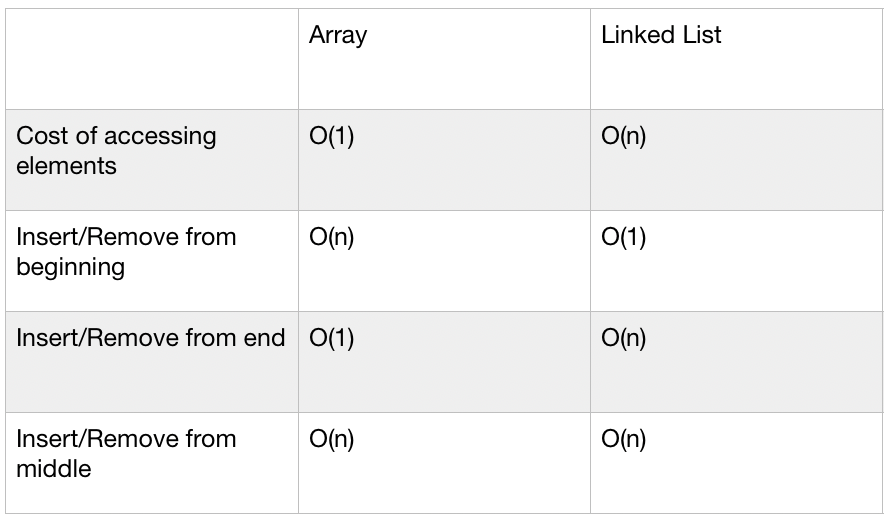
So usually stacks are important in programming because the stack memory is a stack of abstract data types and Queue are important in operating systems.

For example, Android operating system relies extremely heavily on Q abstract data type. You can imagine that a smartphone has to track several events.

For example, the user may touch the screen at a given location, or maybe the user touches the screen at multiple locations. The user may press the home button, the user may say the volume and so on. So there are several reasons the Android operating system has to handle. And these are going to be processed by the central processing unit in a first come, first served manner. And this is exactly what the Queues abstract data type can do.

\*\*\* Data Structures- Binary Search Trees \*\*\*

In Array and LL data structures we came to the conclusion that both of these data structures have several advantages as well as disadvantages.



* So this is exactly what we have been talking about, that if we want to manipulate the first items, then LL are better.
* If we want to manipulate the last items than array data structures are better.
* And as far as memory complexity is concerned, we have to store several references of LL. But there is no need for extra memory when using array data structures.
* What is absolutely crucial that no matter we are using arrays or LL searching for an arbitrary item, if we do not know, the index of the underlying item takes O(N) linear running time complexity for both data structures.

And basically this is exactly the reason why binary search trees came to be so.

Is it possible to do better than linear running time complexity when searching for an arbitrary item? yes, we can use binary search trees.

So a good question is what if the array data structure is sorted?

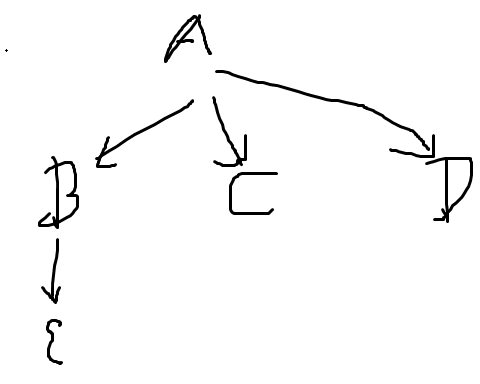
If the array data structure is sorted, then we can search for an arbitrary item with the help of binary search. but basically if we have a one dimensional array of sorted integers, for example, and we are looking for an arbitrary item, then we have to check the middle item and then we have to continue the search operation on the left Sub array or the right Sub array until we find the item we are looking for. So in every iteration, we can discard half of the array. And this is why the final running time complexity will be logarithmic O(log N), which is extremely fast so we logarithmic running times complexity if the array data structure is sorted and this is the main concept behind binary search and binary search trees have the same concept.

We have to store the items in a sorted order, hence we can achieve O(LOG N) logarithmic running time complexity.

**So what are trees in graph theory?**

“A tree is a G(V,E) undirected graph with vertices and e number of edges in which any two vertices are connected by exactly one path. Or equivalently, we can say that a tree is a connected, acyclic, undirected graph. So this is the definition of tree structure in graph theory.

So, for example, this is a typical tree structure.

We have vertices A, B, C, D and E, and we have the edges connecting the given vertices. There is an edge between Vertex A and vertex B, there is a connection between Vertex B and Vertex E, these are undirected edges, as you can see.

What's crucial that we have the so-called root node.

This is the first node of the data structure and it was crucial that we can access the root node exclusively and all other nodes can be accessed, starting with the root node.

This is very similar to what we have seen with LL. We have been talking about the first node of the so-called head a node and we have access to the node exclusively.

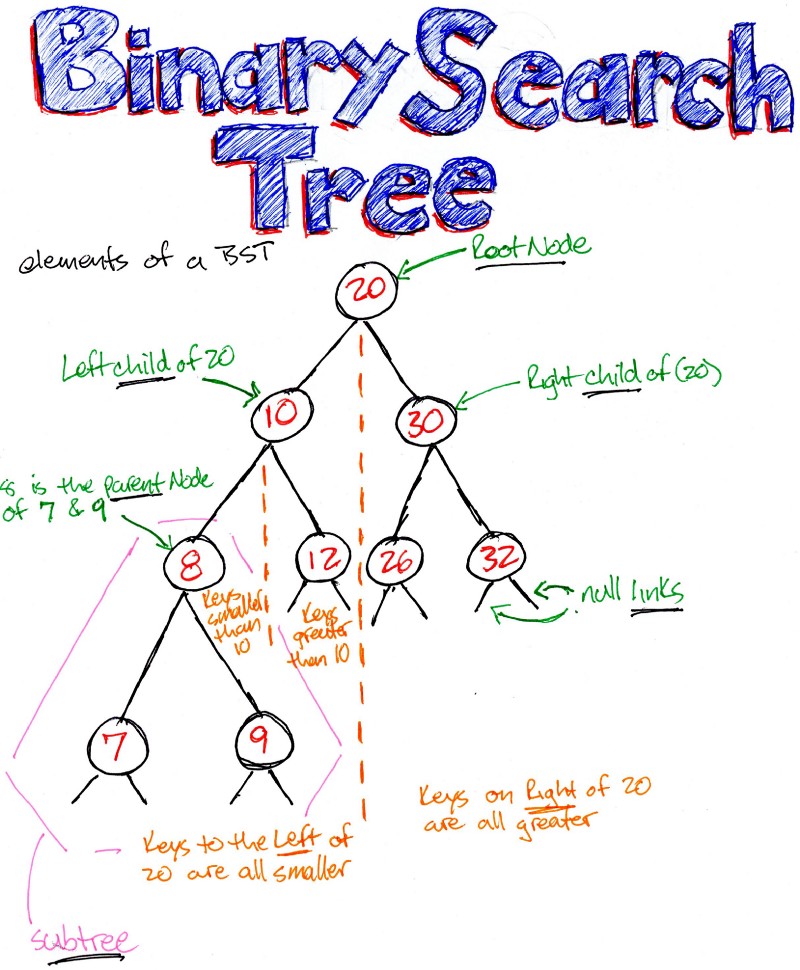
So if we want to access other nodes of the link list, we have to start with the node. And basically this is the case with binary search trees as well.

We have to start with the root node because we have access to the root node exclusively and we may have internal nodes. So internal nodes have a so-called parent and each can have one or more children. And there are the so-called leaf nodes with no children at all.

So Vertex D, the vertex C, vertex E, these are the so-called leaf nodes.

what about binary ST?

Binary search trees can have directed edges, so the parent nodes are pointing in the direction of the children. And it is absolutely crucial that every node in a tree can have at most two children, a so-called left child and the right child. The left child is always smaller than the parent node. the right child is always greater than the parent node, and we can access the root node exclusively and all the other nodes can be accessed starting with the root node.



Because we have been talking about sorted array data structures and we came to the conclusion that we can use binary search and this is why we can achieve a logarithmic running time complexity.

So this is exactly the reason why we have these features, because if we define binary search trees like this, then basically it is a sorted data structure, which means that the parent node is always greater than the left child and always smaller than the right side. So we keep the items in a sorted order.

And why is it good? Because every decision can get rid of half of the data, like with binary search. And this is how we can achieve. O(LOGN)logarithmic running time complexity means that the left child is always smaller, the right child is always greater than the parent node, and it is true for every single node in the tree like structure.

The height parameter of a tree is the number of edges on the longest downward path between the root node and the given leaf node, basically the number of layers in the tree structure.

So first of all, let's consider that how many nodes the given layer is contained. So if we are considering a complete binary search tree, so what does it mean when all the nodes have a left child and right child, then the first layer contains due to the power of zero. So we want to know because it contains the root node exclusively, then the root node may have a left child and the right child.

BST- Search and Insert

how to insert items into a binary search tree and how to search for an arbitrary item.

So, for example, we would like to insert 12 and because the data structure is empty. The first item we insert is going to be the root node of the binary search tree. So we have a root node with value 12.

what if we would like to insert number four? we have to start with the root node because we have access to the root node exclusively. We have to check that whether the value in the actual node we are considering is greater or smaller than the value we would like to insert.

You may pose the question of why? Because this is the topology of binary search trees left children are smaller than the actual node and the right children are greater than the actual node. 4 and 12

So if we have value 4, we would like to insert to the left because the left child is smaller than the parent 12. So because there is no left child, we create a new node in the binary search tree with value for.

we know for certain that the left child is smaller than the parent. So we can say that the left subtree contains items that are smaller than the parent node. So what do we have to do if we want to find the minimum again?

We have to start with the root node and we have to consider the left children until we come to the conclusion that the actual node doesn't have any left child. And we can come to the conclusion that this is the smallest value in the binary ST. And as you can see, it is fine because one is the smallest item in the binary search tree. So the minimum item in the binary search tree is the leftmost item in the tree like structure. So basically we can come to the conclusion that if we use these features that the left child is smaller than the parent know that the right child is greater than the parent node. Then we store the items in sorted order.

And why is it good?

Because in every iteration we can discard half of the original dataset. Hence we can end up with a logarithmic running time complexity. But it is true. If the binary search is balanced, even the binary search tree is imbalanced.

We should keep the health of the tree at a minimum because it means that the given tree like structure will be balanced. And if the tree like structure is balanced, then we can discard half of the data structure in every iteration, which means although LogN logarithmic running time complexity, if the tree gets imbalanced, then this logarithmic running time complexity can be reduced to even linear running time complexity.

DELETE:

**Removing a node**

Remove operation on binary search tree is more complicated, than add and search. Basically, in can be divided into two stages:

* search for a node to remove.
* if the node is found, run remove algorithm.

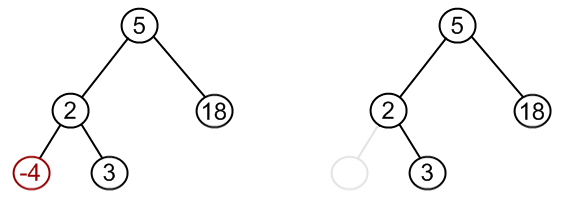
**Remove algorithm in detail**

First stage is identical to [algorithm for lookup](https://www.algolist.net/Data_structures/Binary_search_tree/Lookup), except we should track the parent of the current node. Second part is more tricky. There are three cases, which are described below.

1. Node to be removed has no children.

This case is quite simple. Algorithm sets corresponding link of the parent to NULL and disposes the node.

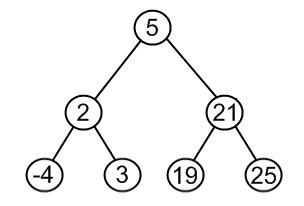
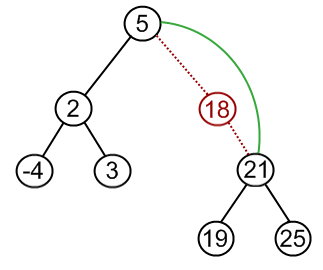
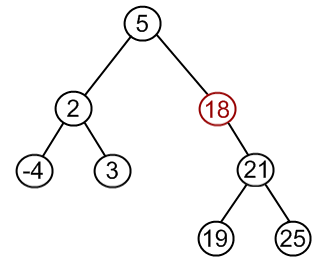
**Example.**Remove -4 from a BST.



1. Node to be removed has one child.

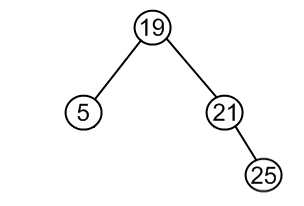
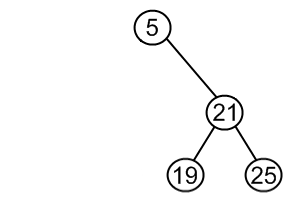
It this case, node is cut from the tree and algorithm links single child (with it's subtree) directly to the parent of the removed node.

**Example.**Remove 18 from a BST.



1. Node to be removed has two children.

This is the most complex case. To solve it, let us see one useful BST property first. We are going to use the idea, that the same set of values may be represented as different binary-search trees. For example those BSTs:

  
contains the same values {5, 19, 21, 25}. To transform first tree into second one, we can do following:

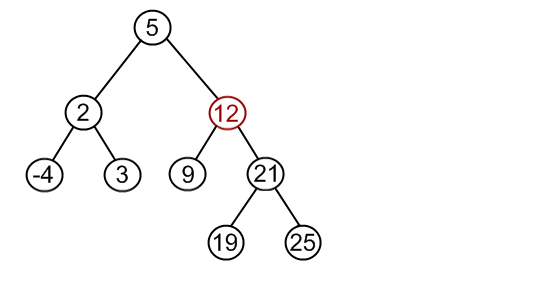
* + choose minimum element from the right subtree (19 in the example);
  + replace 5 by 19;
  + hang 5 as a left child.

The same approach can be utilized to remove a node, which has two children:

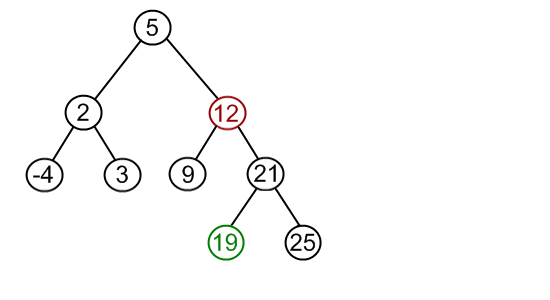
* + find a minimum value in the right subtree;
  + replace value of the node to be removed with found minimum. Now, right subtree contains a duplicate!
  + apply remove to the right subtree to remove a duplicate.

Notice, that the node with minimum value has no left child and, therefore, it's removal may result in first or second cases only.

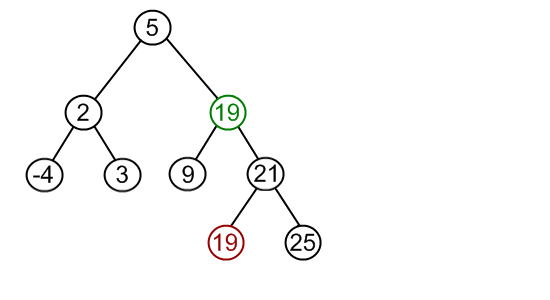
**Example.**Remove 12 from a BST.



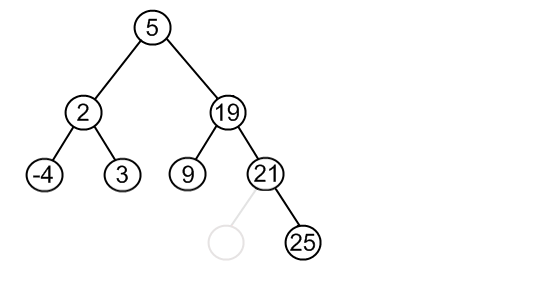
Find minimum element in the right subtree of the node to be removed. In current example it is 19.



Replace 12 with 19. Notice, that only values are replaced, not nodes. Now we have two nodes with the same value.



Remove 19 from the left subtree.



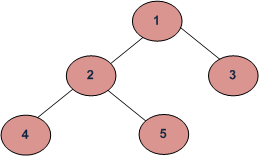
what is the successor, the smallest item in the right subtree is called the successor.

What about the so-called predecessor? The predecessor is the largest item in the left subtree if we consider the sorted

Binary Search Tree Traversal

search for related traversal, so tree traversal means visiting every single node of the binary ST exactly once and because we visit every single node exactly once. This is why this operation has ordered (N) linear running time complexity because we have to consider all the items.

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Depth First Traversals:   
(a) Inorder (Left, Root, Right) : 4 2 5 1 3   
(b) Preorder (Root, Left, Right) : 1 2 4 5 3   
(c) Postorder (Left, Right, Root) : 4 5 2 3 1  
Breadth-First or Level Order Traversal: 1 2 3 4 5

**Inorder Traversal (**[**Practice**](https://practice.geeksforgeeks.org/problems/inorder-traversal/1)**):**

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)

2. Visit the root.

3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of Inorder   
In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal s reversed can be used.   
Example: In order traversal for the above-given figure is 4 2 5 1 3.

**Preorder Traversal (**[**Practice**](https://practice.geeksforgeeks.org/problems/preorder-traversal/1)**):**

Algorithm Preorder(tree)

1. Visit the root.

2. Traverse the left subtree, i.e., call Preorder(left-subtree)

3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses of Preorder   
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on an expression tree. Please see <http://en.wikipedia.org/wiki/Polish_notation> know why prefix expressions are useful.   
Example: Preorder traversal for the above-given figure is 1 2 4 5 3.

**Postorder Traversal (**[**Practice**](https://practice.geeksforgeeks.org/problems/postorder-traversal/1)**):**

Algorithm Postorder(tree)

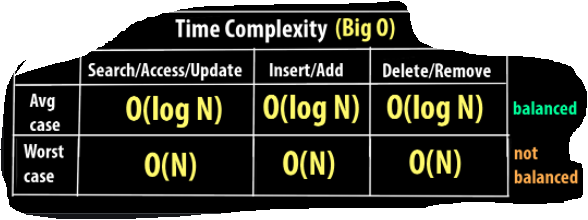
1. Traverse the left subtree, i.e., call Postorder(left-subtree)

2. Traverse the right subtree, i.e., call Postorder(right-subtree)

3. Visit the root.

Uses of Postorder   
Postorder traversal is used to delete the tree. Please see [the question for the deletion of a tree](https://www.geeksforgeeks.org/write-a-c-program-to-delete-a-tree/)for details. Postorder traversal is also useful to get the postfix expression of an expression tree.

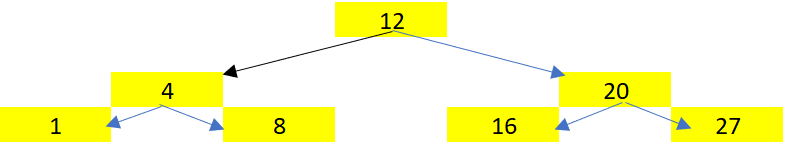
Running Time Complexity



Stack Memory Visualization

visualize the stack of memory as far as the recursive implementations are concerned, because I guess that the most crucial parts in binary search trees and other data structures is when we use recursion.

So, for example, if we have a tree like structure like this, as you can see, it is a standard binary search tree where the left children are smaller than the parents nodes and the right children are greater than the parents node.



So, for example, if we want to find the maximum in a binary ST like this, , we have been talking about that the rightmost item is the largest value in the binary ST.

But how to find it with the help of recursion?

f we know that the given node a child is not a null there, we have to call the function recursively on the right child. But let's take a look at the stack memory and it is going to be clear how it works under the hood.

So, as usual, with binary search trees, we have to start with the root node because we have access

to the root node exclusively. So we call the maximum function on the root node and the root node has the value 12. So this is why this function call will be pushed onto the stack memory. So this is why we have this stack frame maximum with value 12. So we call this function.

So what's going to happen?

The programming language is going to check whether the array child is a null or not. , there is a value the right child. So the programming language will call this maximum function recursively and then it is going to return with the value.

So we call this function recursively.

What does it mean? That there's going to be another function called we value 20 because 20 is the right child of the actual node. So we hope to value 20. And because we call the function recursively, we start again. We check whether the right child over the actual node is not. And , it is not an because it is the value. Twenty seven. So again, we call this function recursively with the right child. So this is why another stack frame is going to be inserted onto the stack memory with value. Twenty seven.

OK, and because we call the function recursively we start again. in this case there is no child for twenty seven so the right child is a null.

Trees Applications:

OS uses almost all the important data structure

* 1. trees are extremely powerful if we want to represent hierarchical data (file system)
     + operating system
     + game trees(chess and tic-toc)
     + machine learning: we like decision trees and boosting uses very simple tree structures

Balanced binary search trees or AVL trees

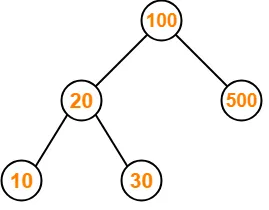
Arrays, LL and binary searches as the crucial data structures. LL and arrays are concerned, there are several advantages and disadvantages for both data structures.

If we want to manipulate the beginning of the data structure, then LL are extremely powerful.

If we want to manipulate the end of the data structure, then arrays are better.

The main problem is that if we want to search for an arbitrary item, then we have no other option but to use linear search in O(N) and of course we can do better with the help of binary search trees.

So we came to the conclusion that if we construct a tree like structure like this out of the data,it is called a binary search then we can achieve Olog(n) running time complexity, the left child is always smaller than the parent node.



The right child is always greater than the parents.So it is quite an easy data structure in the sense that it is quite easy to understand the logic behind binary searches.

We would like to store the items in a sorted order. why?

Because if we have a sorted data structure, then we can use the principle of binary search.

And this is exactly why we can achieve, Olog(N) logarithmic running time complexity.

But of course, as usual, there are some disadvantages for the binary search tree.

So on average, binary searches are working fine and all the operations have already Olog(N) logarithmic running time complexity.

But in worst case scenario, we end up with the LL with linear running time complexities.

Ex: 12->4>3>2 the operations will have O(N) linear running time complexities.

So we came to the conclusion that a binary search can be a balance tree or an imbalance three.

We have to make sure that the binary search tree is balanced because this is when the operations will have, O(LogN) logarithmic running time complexity.

how to make sure that a binary search tree is balanced? We can introduce a new parameter, the so-called height parameter of the tree, and we have to make sure that we keep the height of the tree at a minimum.

**Why AVL Trees?**   
Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take O(h) time where h is the height of the BST. The cost of these operations may become O(n) for a skewed Binary tree. If we make sure that height of the tree remains O(Logn) after every insertion and deletion, then we can guarantee an upper bound of O(Logn) for all these operations. The height of an AVL tree is always O(Logn) where n is the number of nodes in the tree

So every tree is another data structure. the aim is to manipulate data in an efficient manner.

Whether we would like to insert a new item or we would like to find a given item, we would like to

remove a given item and so on.

* It is a balanced DS was invented back in 1962 by two Russian computer scientists Adelson-velsky and Landis (AVL).
* This data structure is very, very similar to binary search trees, but it is a so-called balanced binary search tree, which means that although O(logN)logarithmic running time complexity is guaranteed.
* the running time of binary search tree depends on the h Height of the binary search tree
* in an AVL tree, the Height h parameters of the two child subtrees of any node differ by at most one.
* AVL trees are faster than red black trees because they are more rigidly balanced but needs more work.
* And operating systems relies heavily on the balance.

Both [red-black trees](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree) and [AVL trees](https://en.wikipedia.org/wiki/AVL_tree) are the most commonly used [balanced binary search trees](https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree) and they support insertion, deletion and look-up in guaranteed O(logN) time. However, there are following points of comparison between the two:

* AVL trees are more rigidly balanced and hence provide faster look-ups. Thus for a look-up intensive task use an AVL tree.
* For an insert intensive tasks, use a Red-Black tree.
* AVL trees store the balance factor at each node. This takes O(N) extra space. However, if we know that the keys that will be inserted in the tree will always be greater than zero, we can use the sign bit of the keys to store the colour information of a red-black tree. Thus, in such cases red-black tree takes no extra space.

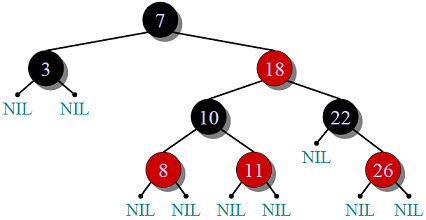
What are the application of Red black tree?

Red-black trees are more general purpose. They do relatively well on add, remove, and look-up but AVL trees have faster look-ups at the cost of slower add/remove. Red-black tree is used in the following:

* Java: [java.util.TreeMap](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/TreeSet.html), [java.util.TreeSet](https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/TreeSet.html)
* C++ STL (in most implementations): map, multimap, multiset
* Linux kernel: completely fair scheduler, linux/rbtree.h

As far as AVL trees are concerned, these data structures are rigidly balanced. That is why O(logN) running time complexity is guaranteed.

***Red Black Tree***:



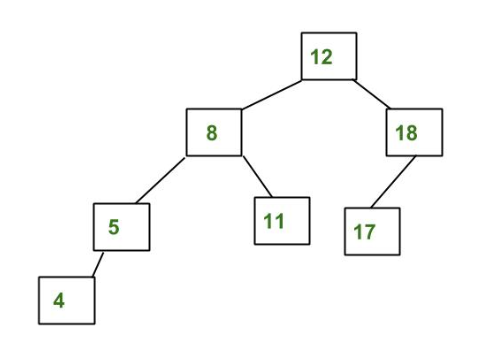
***Properties***:

1. Self-Balancing is provided by painting each node with one two colors(Red or Black).
2. When Tree is modified, new tree is subsequently rearranged and repainted.
3. It requires 1 bit of color information for each node in tree.

***Constraints* maintained by Red Black Tree**:

1. Root is always black.
2. All NULL leaves are black, both children of red node are black.
3. Every simple path from a given node to any of its descendant leaves contains the same number of black  
   nodes.
4. Path form root to farthest leaf is no more than twice as long as path from root to nearest leaf.

**AVL(Adelson-Velskii and Landis) Tree**

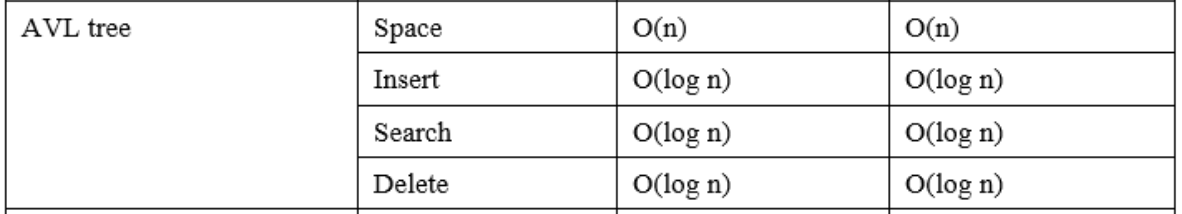


***Properties***:

1. Height difference of left and right subtree of node should be less than 2.
2. Re-balancing is done when heights of two child subtrees of a node differ by more than one.
3. Faster retrievals as strictly balanced.

***Difference*:**

1. AVL trees provide **faster lookups** than Red Black Trees because they are more strictly balanced.
2. Red Black Trees provide **faster insertion and removal** operations than AVL trees as fewer rotations are done due to relatively relaxed balancing.
3. AVL trees store **balance factors or heights** with each node, thus requires storage for an integer per node whereas Red Black Tree requires only 1 bit of information per node.
4. Red Black Trees are used in most of the language libraries like [map](https://www.geeksforgeeks.org/map-associative-containers-the-c-standard-template-library-stl/), [multimap](https://www.geeksforgeeks.org/multimap-associative-containers-the-c-standard-template-library-stl/), [multiset](https://www.geeksforgeeks.org/multiset-in-cpp-stl/) in C++ whereas AVL trees are used in **databases** where faster retrievals are required.



**AVL trees - Height**

If the tree like structure is imbalanced, then in a worst case, we can end up with a long list with linear running time complexities.

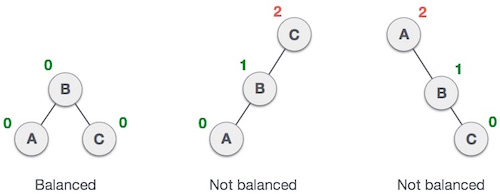
So this is why we have to track whether a given binary search tree is imbalanced or not.And this is why we have to assign the Height parameter to every single node in the binary search tree.

“The height of a tree is the number of edges on the longest downward path between the root node and the leaf node. the number of layers the binary search tree contains, this is the height of the binary search tree.”

Height = max(left child’s height , right child’s height) + 1

AVL tree rotation:

Here we see that the first tree is balanced and the next two trees are not balanced −



In the second tree, the left subtree of **C** has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of **A** has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

***BalanceFactor*** = height(left-sutree) − height(right-sutree)

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

AVL Rotations

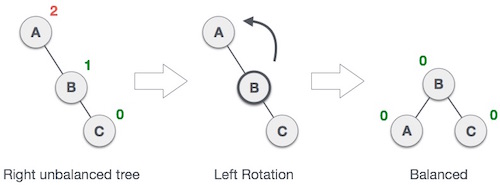
To balance itself, an AVL tree may perform the following four kinds of rotations −

* Left rotation
* Right rotation
* Left-Right rotation
* Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

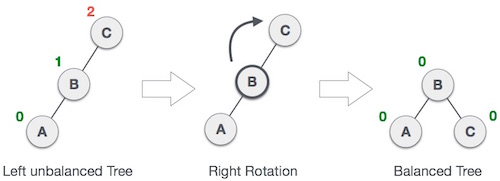
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation −



In our example, node **A** has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making **A** the left-subtree of B.

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.



As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

|  |  |
| --- | --- |
| **State** | **Action** |
| Right Rotation | A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation. |
| Left Rotation | We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**. |
| Left Rotation | Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree. |
| Right Rotation | We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree. |
| Balanced Avl Tree | The tree is now balanced. |

Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

|  |  |
| --- | --- |
| **State** | **Action** |
| Left Subtree of Right Subtree | A node has been inserted into the left subtree of the right subtree. This makes **A**, an unbalanced node with balance factor 2. |
| Subtree Right Rotation | First, we perform the right rotation along **C** node, making **C** the right subtree of its own left subtree **B**. Now, **B** becomes the right subtree of **A**. |
| Right Unbalanced Tree | Node **A** is still unbalanced because of the right subtree of its right subtree and requires a left rotation. |
| Left Rotation | A left rotation is performed by making **B** the new root node of the subtree. **A** becomes the left subtree of its right subtree **B**. |
| Balanced AVL Tree | The tree is now balanced. |

Application of AVL Trees:

1. binary searches and balance, binary search trees can be used in game engines.

So almost all the game engines use binary search trees to determine which objects should be rendered in the game world.

By the way, this approach was first used back in 1993 during the implementation of Doom. So first of all, balanced binary search trees can be used in a game engines.

1. we can use binary search within compilers.

So it is absolutely crucial in programming languages to be able to parse syntax expressions such as loops and statements. And this is why compilers use a special a binary search tree, a so-called syntax tree.

So a syntax tree is very, very similar to a binary ST. Certainly there are parents and children in relationships. So basically it is a tree like structure and it helps a lot for compilers.

1. AVLsort we can use a balanced binary search trees in order to sort an underlying data set so we can use a balanced tree data structure to sort items efficiently in O(NlogN) linear running time complexity.

There are divide and conquer approaches such as quicksort and Merge sort with O(NLOGN) running time complexity and we can achieve the same with the sort.

For example, if we have an integer, we would like to sort them.

First of all, we have to create an AVL tree, so we have to insert the items one by one into an AVL tree.

It can be done in O(N) running time complexity because we have N items and for every single

item it takes O(NLOHN)+ O(N) == O(NLOGN)

quicksort is regarded as the best sorting algorithm possible.

1. Databases :we can construct databases with the help of balance binary search trees.

Usually we have to make sure that there are going to be several lookups or search operations because if there are several remove operations or insertions, then every tree is not the best option possible because of the huge number of rotations.

So AVL trees make several rotations and this is one of the disadvantages of AVL tress.This is why red the black trees are regarded as a bit more efficient data structure.But by the way, as far as databases are concerned, the best approach is to use B trees.

\*\*\*\* SUBSRING SEARCH ALGORITHMS \*\*\*\*

Brute- force Search

**Naive Substring Search**

* we would like to construct an algorithm thats capable of finding a **P** pattern in a given **S** string ortext
* **brute-force search** is the naive approach
* keep iterating through the text and if there is a mismatch we shift the pattern one step to the right

In this case, we have to a get A A G and we are looking for a given pattern



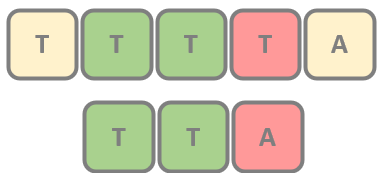
* not so efficient especially when there are lots of matching prefixes
* main problem is that we need backup for every mismatch
* if there is a mismatch we jump back to the next character
* there are lot of comparisons – **O(NM)** where **N** is the length of text and **M** is the length of the pattern
* **linear running time** would be better

OK, so for example, we have the original text. This is a test and we are looking for a given pattern.

So this is the pattern we would like to find in the original text.



*the* ***worst-case scenario*** *occurs when there are several matching* ***prefixes***



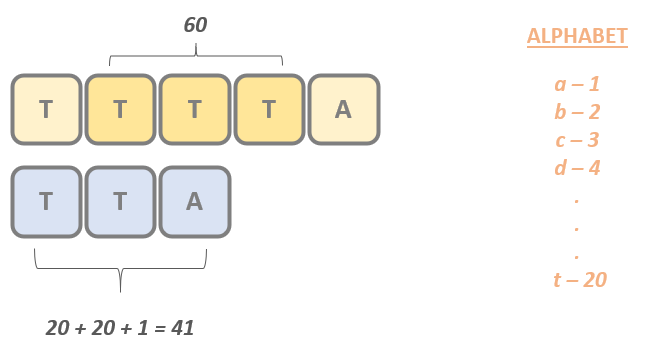
**Rabin-Karp Substring Search**

* it was first constructed in **1987** by **Richard Karp** and **Michael Rabin**
* we have considered brute-force substring search algorithm
* the problem that is have to make **too many comparisons** when there are matching prefixes
* it has **O(M+N)** running time but the worst-case is still **O(MN)**
* **IS IT POSSIBLE TO COMPARE THE SUBSTRINGS IN O(1)?**
* the aim is to compare the pattern and the region of the text with a single test – in **O(1)** constant running time

 in the standard approach we have to compare **3** characters (size of pattern **m=3**)

*this is how we* ***transform multiple characters into a single value*** *(integer) and we just have to compare the integers*

what do we have to do as usual with hash functions?



We are going to transform the letters into memory representations, and we can use the ASCII value, for example, or we can use just the simple representation in our example, just for demonstration purposes.

We are going to use integer one for letter A that integer two four letter B, then three four letters

C and so on.

D has the integer representation 20, so 20 plus 20 because we have another letter D and finally we have a letter A. So 20 plus twenty plus one is equal to forty one. So it means that we have managed to transform the whole pattern into a single integer, which means that forty one is going to represent the pattern with these three characters.

why do we have to deal with the text?

We have to consider as many letters in the text as the length of the pattern. So in this case, we have to consider are just three characters.

And of course, we have to do the same.So because we have T, T T and the T has the integer representation 20, this is why 20 plus 20 plus 20 z close to 60, which means that the substring has the integer representation. 60.

So if we want to compare the better with the substring, then we have to compare 41 and 60 and we can come to the conclusion that of course, it is a mismatch because 60 is not equal to 41, so we keep going.

So this is the basic idea behind rabian carp algorithm that we are going to use hash functions in order to convert multiple characters or basically soft strings into integer representations. And then we just have to compare the integers as far as these letters are concerned to has the value

20 plus twenty plus one so forty one, which means that in this case, forty one is equal to forty.

So even the hashes are matching, then we have to compare the letters on a one by one basis, the letters are matching. Another three letters are matching and finally the letters a-r matching.

So if we come to the conclusion that the hash values are the same, then we have to do exactly the same as we have seen with brute force search. We have to check the letters on a one by one basis.

* so we can skip several comparison operations if we rely on **hashing**
* we have to compute the **h(p)** hash value of the pattern in advance in **O(m)** running time – **m** is the length of the pattern
* if the hash values are matching then we have to use **standard character comparisons** one by one (maybe there is a **spurious hit**)

disadvantages of hash functions.

If a hash function is not perfect, then there may be some problems.So if the hash values are matching, then we have to use standard character comparisons one by one.

This is what we have been talking about, and this is exactly the same as we have seen with the brute force approach.

And it is absolutely crucial that there may be spurious hits or spurious states.

What does it mean exactly that?

Because hash functions are not perfect in real life, it may happen that we get the same hash value

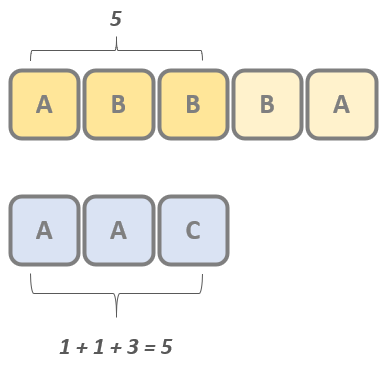
for different letters. So it may happen that there is a mismatch of the characters, despite the fact that the hash values are the same and these are called spurious hits.

So for example, if we had this example, we have a A and C as far as the pattern is concerned and we

have the taxed a BBB, a what do we have to do?

Of course, we have to apply the same approach.

As we have discussed previously, a has the integer representation of one. A has the integer representation, one C has the integer representation three. So one plus one plus three is equal to five.



What about the substring string?

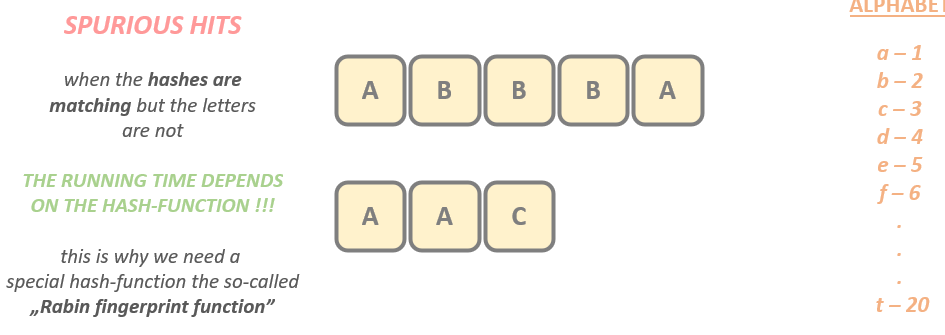
In the original text, we have a B and be a has the integer representation one b has the integer representation to you.

Then we have another B letter to be the value to you. So one plus two plus two is equals to five again. So based on the hash values, we can come to the conclusion that it is a match in the sense that we have managed to find the pattern in the text.

But if we compare the given letters that will be come to the conclusion that, OK, it is a mismatch. So these are the so-called spurious states, which means that the hash values are the same, but the letters are not matching.

For example, we can come to the conclusion that B, B and B has the hash value six.

So the running time depends extremely heavily on the hash function. And this is why we need a special hash function.



The so-called rabian fingerprint function if we use the fingerprint function.Then, of course, there may be spurious hits as well, but with extremely low probability.

So in the main rabian Karp algorithm is going to have, although AM plus and linear running time complexity.

The **Rabin fingerprint function** uses a **polynomial** – this is how the spurious hits can be eliminated with high probability

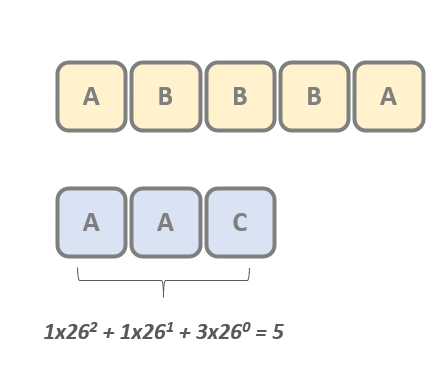
So what about the fingerprint function?

The fingerprint function uses a so-called polynomial. This is how the spurious heaths can be eliminated.

Of course, a polynomial is something like this.

**f(x) = m0 + m1 x + ... + mn-1xn-1**

pattern like this V letter A, A and C in the previous approach, we just consider the given letters.



We checked the integer representations and we summed up the values 20 plus twenty plus one is equal to 41.

to the number of letters in the pattern minus swan, you may pose the question why?

Because it has something to do with this polynomial function, as you can see, and the minus one is

the last term here and is the number of letters in the pattern and is equal to three three minus one

is equals two to you.

So this is why the first term one times twenty six to the power of two plus we have the integer representation

of eight again.

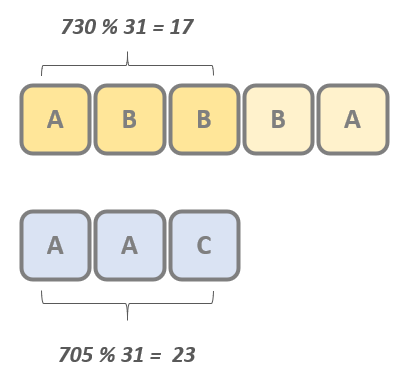
So this is why one times twenty six to the power of one, then we have the last term three because letter

C has the integer representation three and twenty six to the power of zero.

So as you can see, it is a polynomial because the first theorems 26 to the power of zero, then 26,

the power of one, then 26, the power of two and so on, so that fingerprint function uses a polynomial

under the hood.

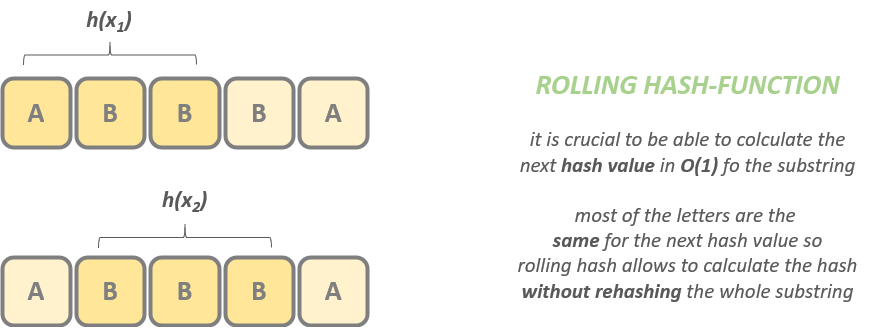


why is it good?

Because the final running time complexity will be, although am a plus and so linear running time complexity. so using a good hash function can effectively eliminate spurious hit. So this is why Arabian fingerprint function is extremely powerful.

Of course, we had another problem that if we have calculated the hash function for a stop string of

the original text, then of course, what do we have to do?



And this is exactly the reason why Arabian being subs things search algorithm can have linear running time complexity because otherwise, if this order was running time complexity is, for example, linear running time complexity, then we end up with brute force search again, and this is exactly what we want to eliminate.

So if we can make sure that we can calculate these hashes in order one constant running time complexity and this is exactly what's happening when dealing with rolling hash functions, then we can achieve linear substring search.

