Lecture 15

تعالوا نفتكر هي ايه كانت ميزة ال binary search treeهي كانت اني اقدر اعمل سيرش فيها بسرعة

المحاضرة دي هنبص على algorithms تانية بنعمل سيرش فيها ببطأ علشان المحاضرة الجاية نعمل مقارنة بينهم كلهم بقى.

هنبص بقى على algorithms الداتا متخزنة جواها في linear data structure زي ال arrayمثلا (الtrees كانت hierarchy data structure)

اول حاجة هنتكلم عليها هي ال **linear search** :

1	7	5	13	29	18

عايز ادور على ال element رقم 13 .. همشي على ال elements واحد واحد و اقارن لحد م اوصل لرقم 13 و بعدها اخرج من ال function بتاعتي..
طب لو كنت بدور على رقم 19 .. هفضل ماشي لحد اخر ال array و مش هلاقيه و بعدها هخرج.. كذلك لو بدور على 100 او 1000 .. هلف في ال array من الاول لحد الاخر و بعدين هخرج

يبقى نقدر نقول ان لو الداتا اللي بدور عليها موجودة هفضل الف لحد م اوصلها .. لكن لو مش موجودة فأنا كده هعمل number of comparisons بيساوي N اللي هو عدد ال elements اللي موجودة جوه ال array او ال size بتاع ال array

- For each item in the list
- if the item's key matches the target,
- stop and report "success"
- Report "failure"

ال key ده المقصود بيه رقم بيعبر عن ال element ده .. زي مثلا م كل واحد له البنش نمبر بتاعه انما الداتا الحقيقية اللي بدور عليها هي مش البنش نمبر .. لكن هي اسم الطالب و درجاته و ..و..

طب نبص بقى كده على ال implementation

```
int linearSearch(int a[], int n, int target)
{
  int i;
  for (i = 0; i < n; i++)
    if (a[i] == target) // key comparison
      return i;
  return -1; // use -1 to indicate failure
}
```

n ده بيعبر عن عدد ال elements جوه ال array و التارجيت هو ال key او ال value اللي بدور عليها.. في الكود ده انا برجع الlocation بتاع ال key اللي بدور عليه
نلاحظ ان مفيش اي constrains على الداتا اللي بحطها في ال array .. يعني مش شرط تكون مترتبة .. لكن لو نفتكر في ال binary search tree هي كانت مبنية على ان الداتا اصلا مترتبة .. وانا بحط اصلا الداتا برتبها (less than goes left, greater than goes right)

هنا الداتا اللي بتجيلي بحطها في اي مكان فاضي و خلاص... دلوقتي بقى عايزين نبص على سرعة الalgorithm ده

- **Speed of an algorithm is measured by counting key comparisons**
- **Best case is 1 comparison**
- **Worst case is n comparisons**
- **Average case is n/2 comparisons**
  - if the target is in the list

نلاحظ ان ال worst case دي هتحصل لما اكون بدور على اخر element في ال array او بدور على حاجة مش موجودة اصلا في ال array.. ال average هنا n/2 بفرض ان ال element اللي بدور عليه موجود جوه ال array

- What if the target we are looking for has only a 50% chance of being in the list?
- The complexity must account for both targets that can be found and targets that cannot.
- For targets that can be found it is n/2
- For targets that cannot be found it is n
- For targets that can be found only 50% of the time it is: 1/2*n/2 (found) + 1/2*n (not found)
- The order of complexity therefore is: n/4 + n/2 = 3n/4

لو بقى الحاجة اللي بدور عليها مطلعتش موجودة في ال list بتاعتي .. دايما هروح ان عدد ال comparisons يكون n فلو عايز احسب ال Average search time :

E (T: value of search time) = P (we have the element) * Excepted time + P (element does not exist) * Max time

نقدر نكتب الكلام ده كده:

E (T) = P (we have the element) * n/2 + P (element does not exist) * n

لان انا عندي هنا two scenarios..الاول ان الحاجة اللي بدور عليها تكون موجوده اصلا في ال array

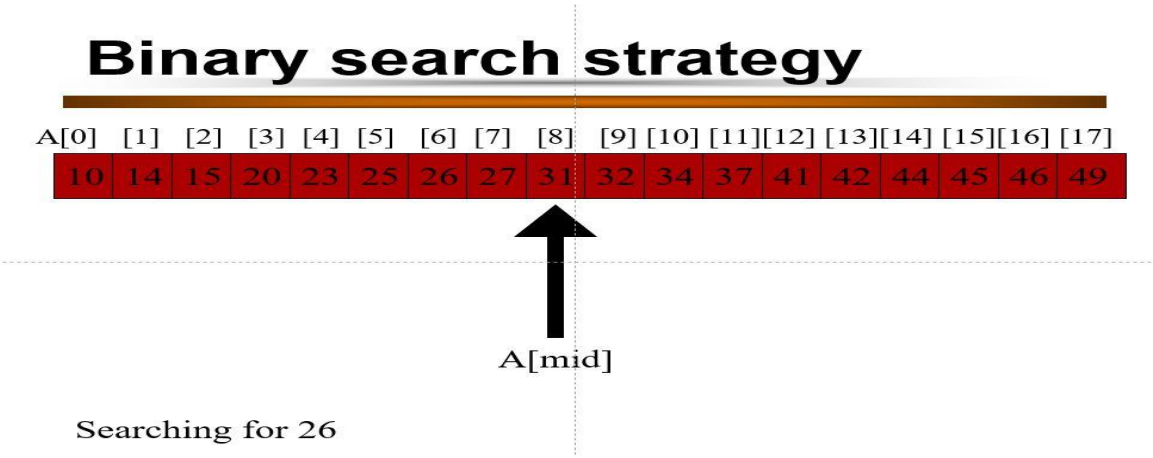
لو كانت احتماليه اني الاقي ال elementهي 50%.. يبقى المعاداة اللي فوق دي هتبقى

E (T) = 0.5 *n/2 + 0.5*n = ¾ *n

و هو ده هيبقى ال order of complexity

((الدكتور عدي 3 سلايدز و قال هيرجعلهم المحاضرة الجاية))

طب نروح نشوف بقى algorithm تاني اسمه **binary search**:



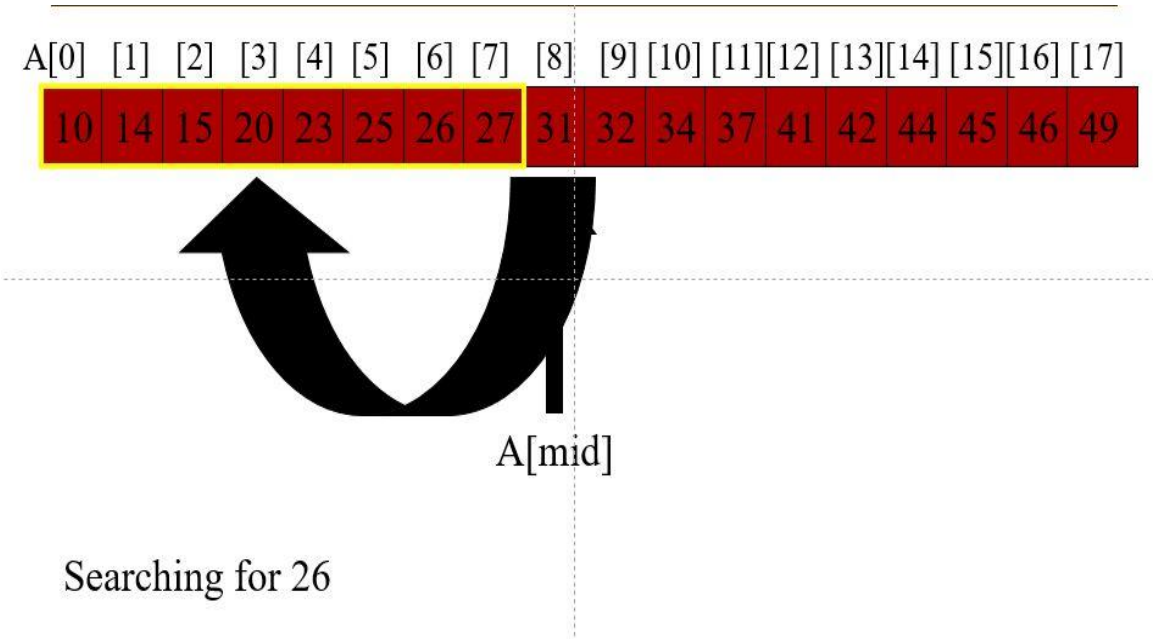
هنا انا و انا بدخل الداتا جوه ال array هرتبها .. يعني وانا بعمل insert هدخل في المكان المظبوط .. زي ال ordered list

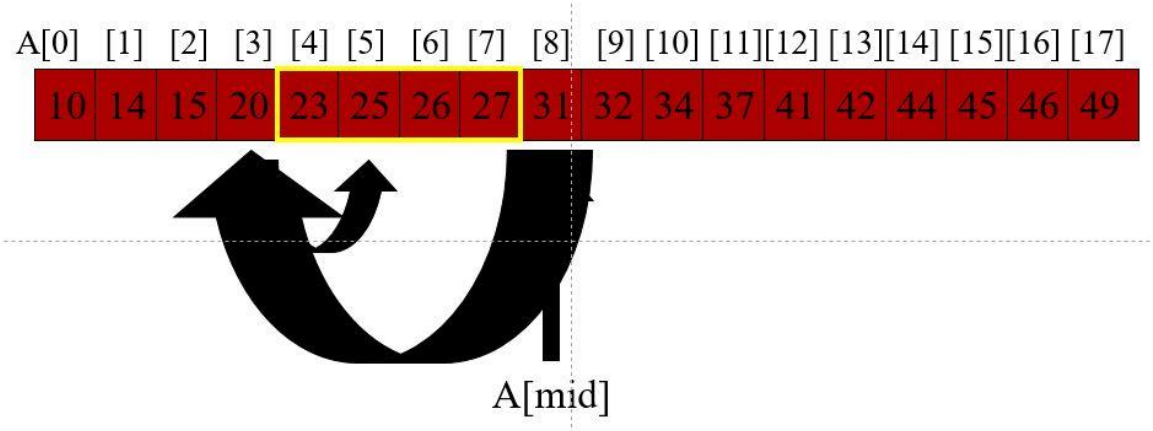
الدكتور قال ممكن نخلي الكلام ده مترتب عن طريق استخدام ordered list و هنعرف نشتغل بيها في ال binary search بس الكلام ده الدكتور قال نفك فيه كده و نشوف هل هو دقيق ولا مش دقيق!!

في ال binary search احنا بندأ ندور من النص.. و اقارن القيمة اللي في النص دي مع القيمة اللي عايز اوصلها .. و طالما ال array مترتب يبقى لو طلعت اكبر من اللي بدور عليها كده اكييد القيمة اللي عايزها لو هي موجودة في ال arrayده هتبقى موجودة في النص اللي على الشمال.

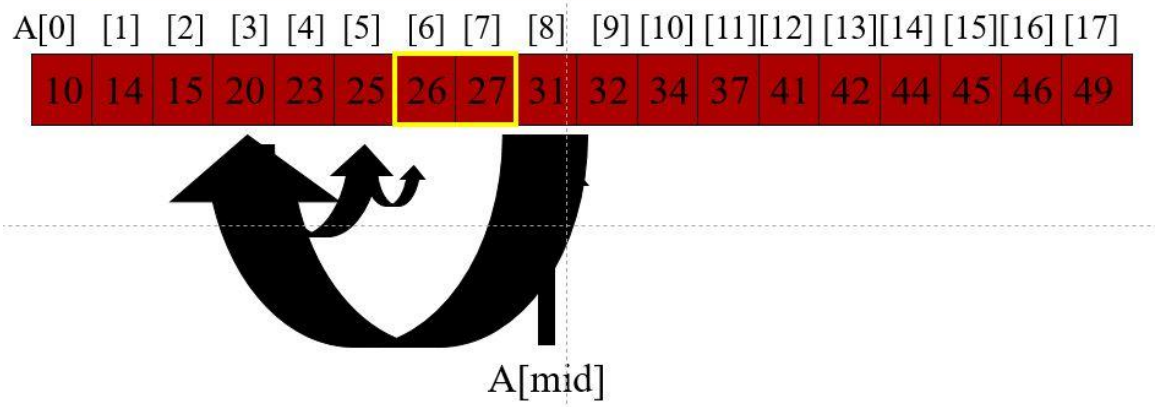
هكرر اللي عملته ده تاني.. دلوقتي انا بقى عندي array جديدة .. البداية بتاعتها 10 و النهاية 27.. هدور فيها بنفس الطريقة .. اني اروح على النص بردو و اقارن.

في ال algorithm بتاع المحاضرة قالك لو عدد ال elements جوه ال array كان even .. هاعتبر ان اللي في النص هو اللي على شمال النص..(شمال ال boarder اللي بيمثل النص)





Searching for 26



Searching for 26

كده انا عملت 4 خطوات لحد م صلت لل target ... لو كنت بدور linear كنت عملت 7 خطوات لاننا بدأنا من الصفر هنا.

كده احنا اهو لما حطينا constrains اكتر و احنا بنحط الداتا ..قدرنا ندور اسرع من ال linear search.. علشان كده الناس بتفضل دايما انها تستخدم ال binary search عن ال linear search

The growth of the base-2 log function

n	n (as power of 2)	$\log_2 n$
16	2^4	4
256	2^8	8
4,096	2^{12}	12
65,536	2^{16}	16
1,048,576	2^{20}	20

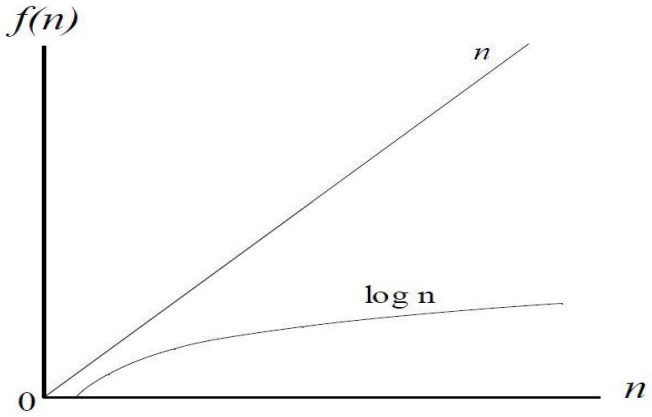
دلوقتي لو حجم ال array كان n.. ف عدد الخطوات اللي بحتاجها علشان اوصل ل element اللي بدور عليه هو proportional ل $\log_2 n$

يعني لو حضرتك عندك array حجمة 16 .. و ال array ده تضاعف و بقى 32 .. عدد الخطوات الزيادة اللي هاخدها علشان اوصل لل target هيزيد خطوة واحدة بس.

ده اللي بندورعليه دايما هو ازاي اخلي ال complexity بتاعتي as function of the size of the problem as fast as possible

انا مش ايز كل م n تزيد .. ال complexity بتاعتي تزيد بشكل كبير هي كمان

Graph illustrating relative growth of linear and logarithmic functions



مثال على كده .. احنا كنا اخدنا DFT بس ال implementation اللي الناس بتستخدمة علشان تعمل ال DFT هو اسمه FFT.. و السبب انهم بيستخدموه هو انه يسخلي ال complexity بتاعه المسألة يبقى $\log_2 n$

نبص بقى على الكود ده كده

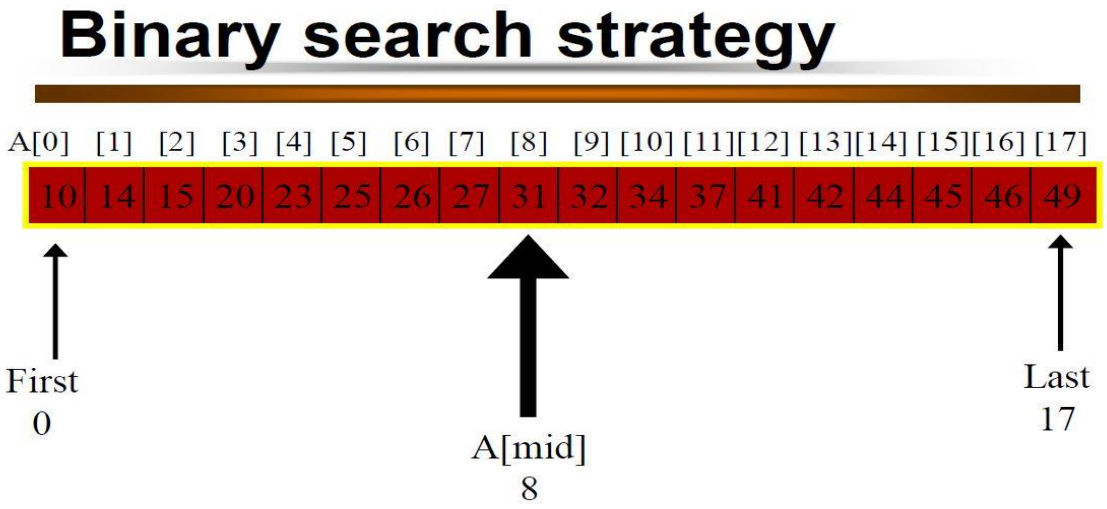
Defective Binary Search

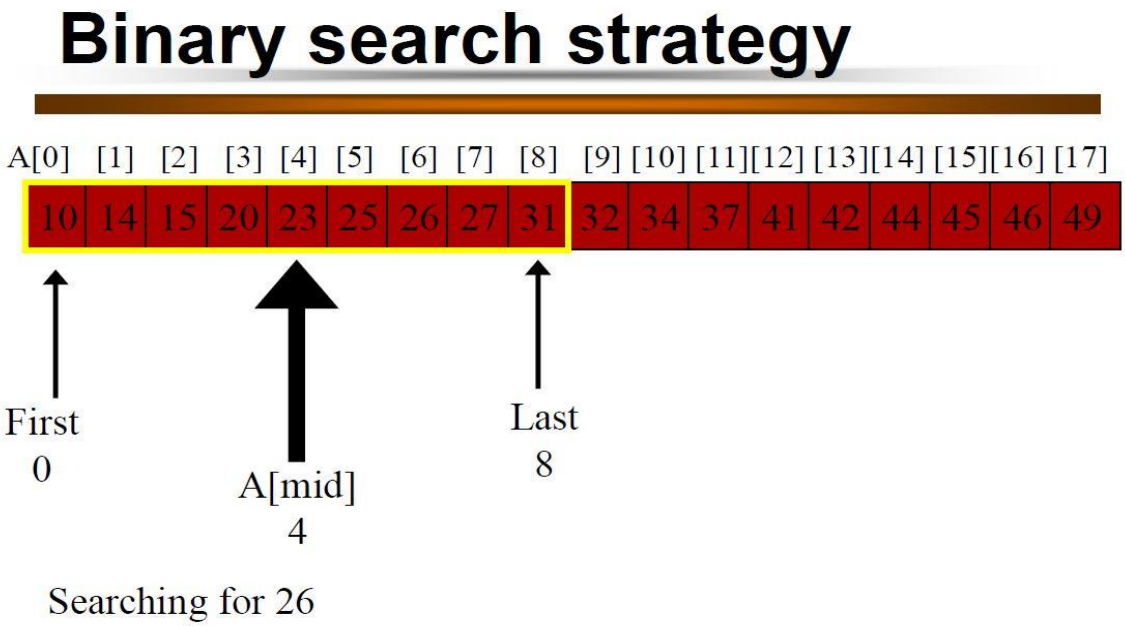
```
int binarySearch(int a[], int n, int target)
{ // Precondition: array a is sorted in ascending order from a[0] to a[n-1]
  int first(0), last(n - 1), int mid;
  while (first <= last) {
    mid = (first + last)/2;
    if (target == a[mid])
      return mid;
   else if (target < a[mid])
      last = mid;
    else // must be that target > a[mid]
      first = mid; }
  return -1; // use -1 to indicate item not found
}
```

السطر ده int first(0) هو هو int first =0 ..و last(n-1) هي هي last = n-1

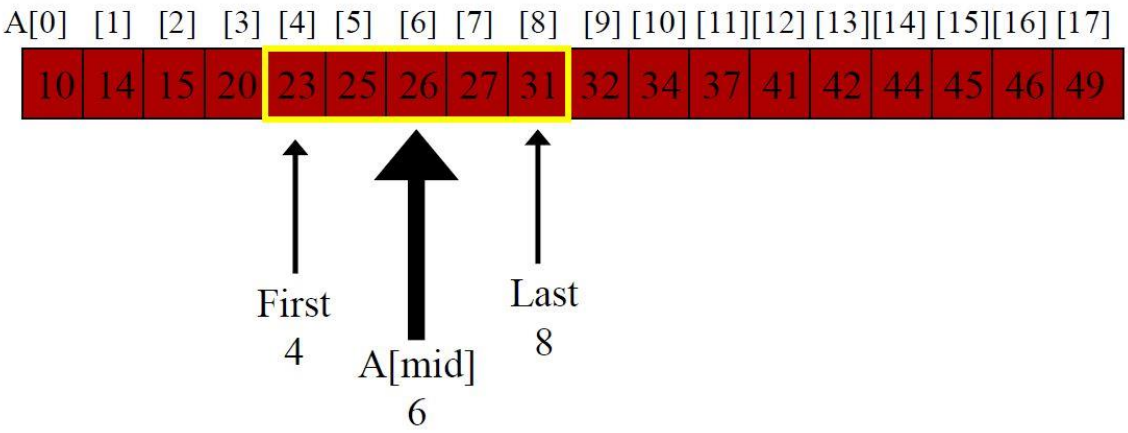
جوه الكود ..لان mid متعرفة انها int فهي will ignore الجزء ال floating اللي هيطلع من القسمة.

الكود هيتنفذ زي م مرسوم في الصو كده:





Binary search strategy



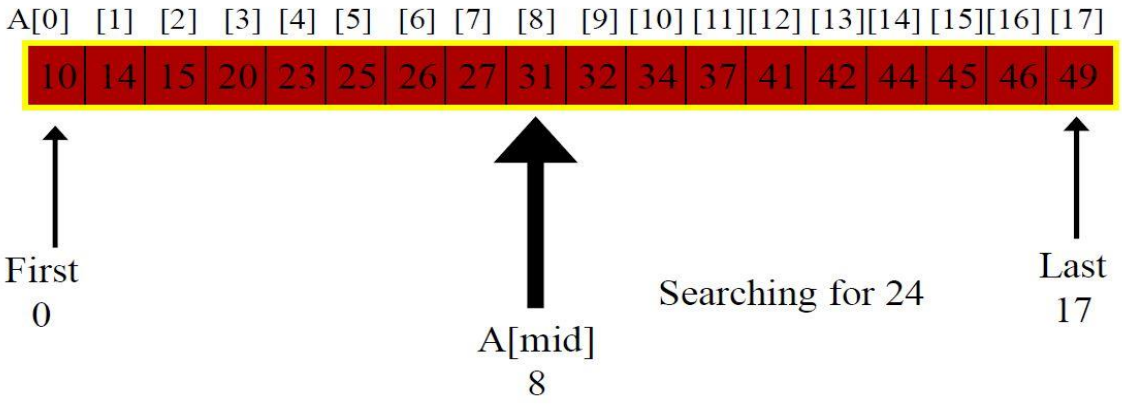
Searching for 26

Problem

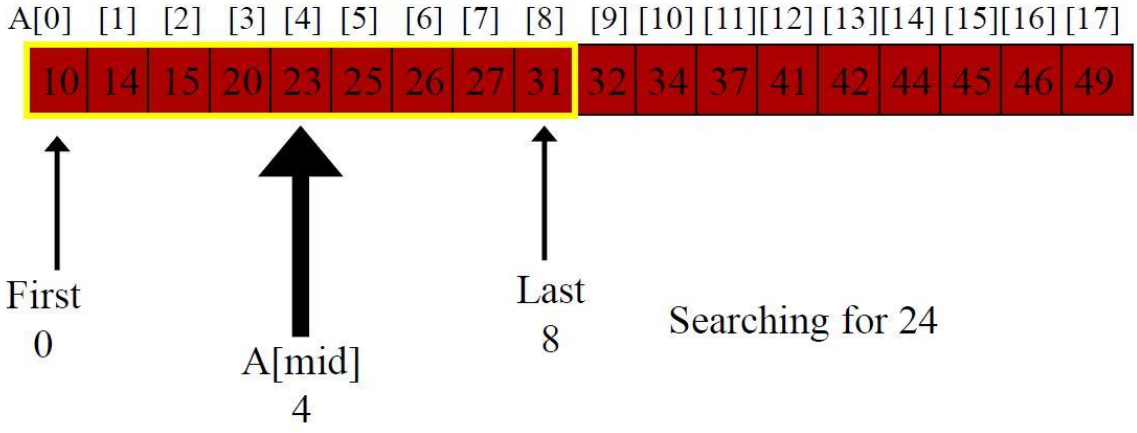
- **This code works fine for finding an item that is contained in the list.**
- **However, it does not work so well if the item is not in the list.**
- **We get an infinite loop in that case.**

الكود ده كده في مشكلة .. تعالى نجرب ندور على قيمة مش موجودة جوه ال Array..تعالوا ندور على 24..هروح بين 23و25 (5 << 10/2 >> 6+4=10).. هروح على 25..الاقيها اكبر من 24..(4>>4.5=9/2>>9=4+5).. هروح على 23 و اقارن بيها الاقيها اصغر ..و هفضل بين الاتنين دول علطول 23و25

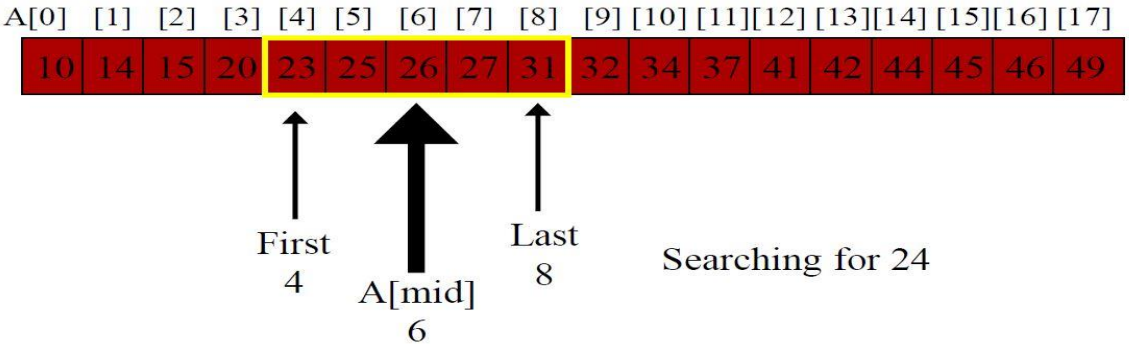
Binary search strategy

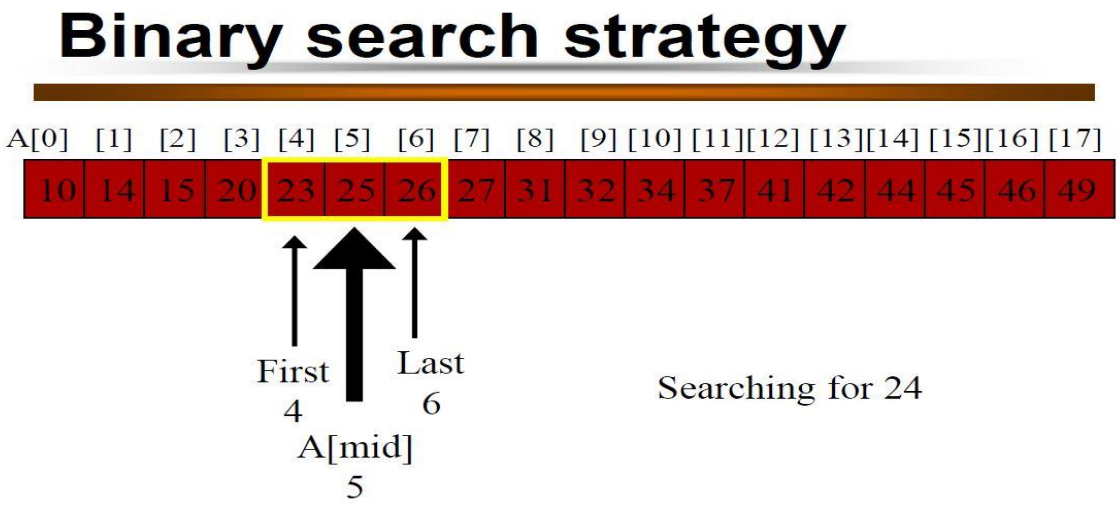


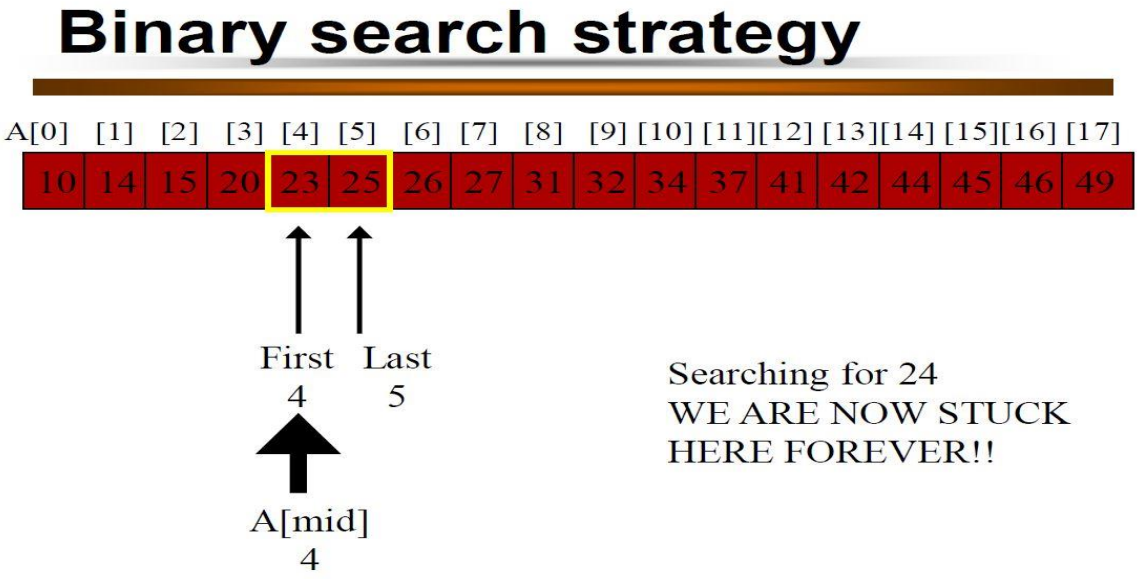
Binary search strategy



Binary search strategy







الحل للمشكلة دي .. ان في ال case اللي هتحرك فيها ال first خلي ال first = mid+1;

..و ان في ال case اللي هتحرك فيها ال last خلي ال last = mid-1;

Verified Binary Search

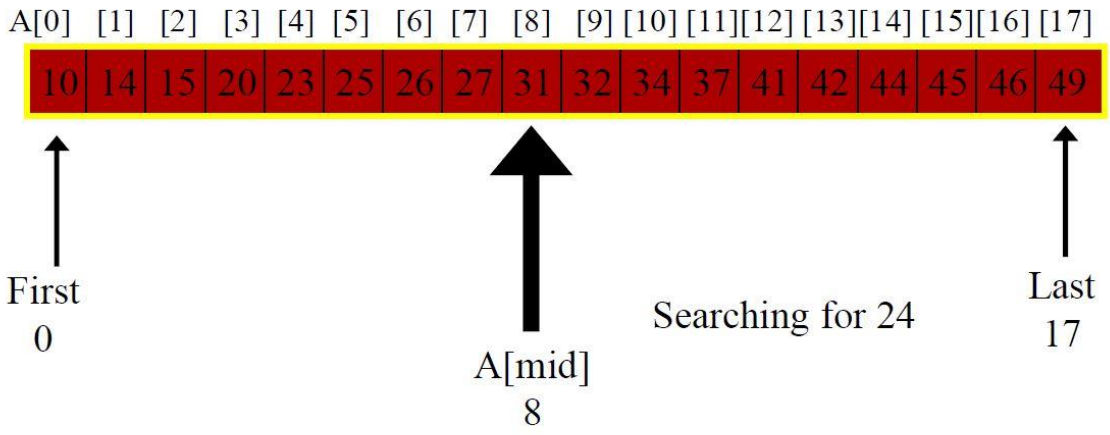
```
int binarySearch(int a[], int n, int target)
{
  // Precondition: array a is sorted in ascending order
  // from a[0] to a[n-1]
  int first(0); int last(n - 1);  int mid;
  while (first <= last) {
mid = (first + last)/2;
 if (target == a[mid])
       return mid;
    else if (target < a[mid])
      last = mid - 1;
    else // must be that target > a[mid]
      first = mid + 1;
  }
  return -1; //use -1 to indicate item not found
}
```

الفكرة اني لما اعمل مقارنة مع ال mid و تطلع مبتساويش القيمة اللي بدور عليها و تكون القيمة اللي بدور عليها اكبر.. ليه بقى اخلي ال first بتاع ال array الجديد اللي هو sub array من القديم يساوي ال mid .. المنطقي ان لو الميد كانت بتساوي القيمة اللي بدور عليها كنت خلاص خرجت .. ف الصح اني اخلي ال first تساوي mid+1

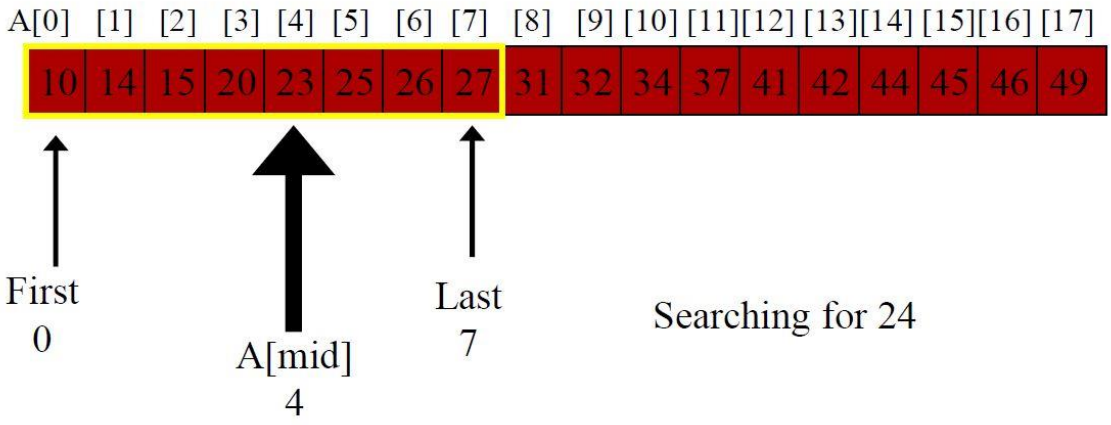
و لو لقيت last اصغر من first بخرج من اللوب علشان كده اللي بدور عليه مش موجود

..الصور الجاية بتوضح الكلام ده بردو:

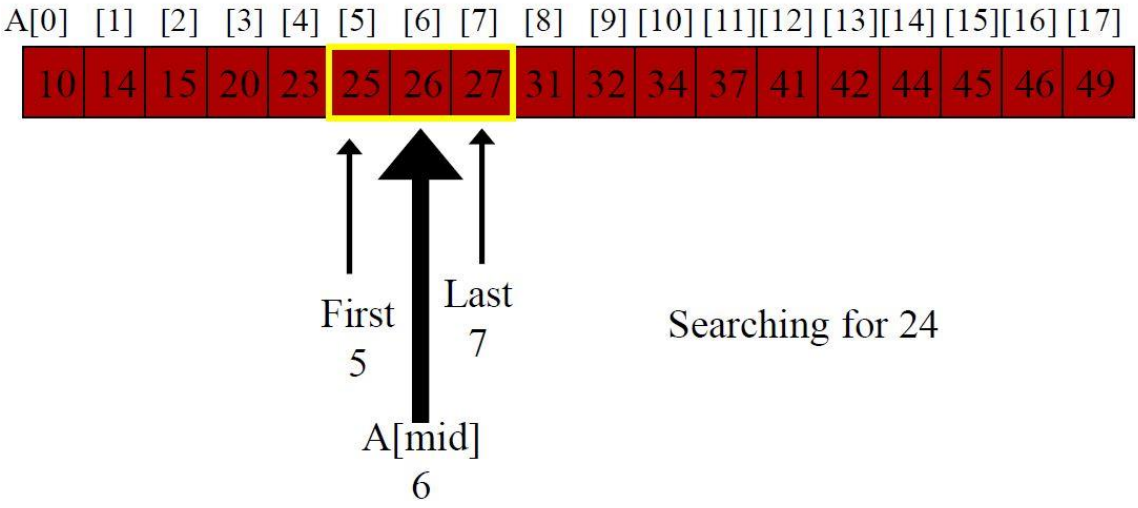
Binary search strategy



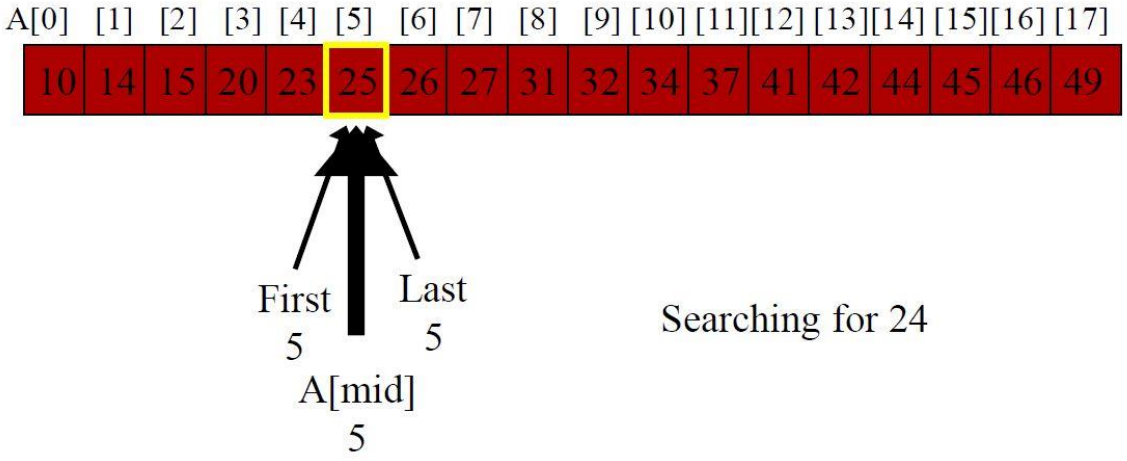
Binary search strategy



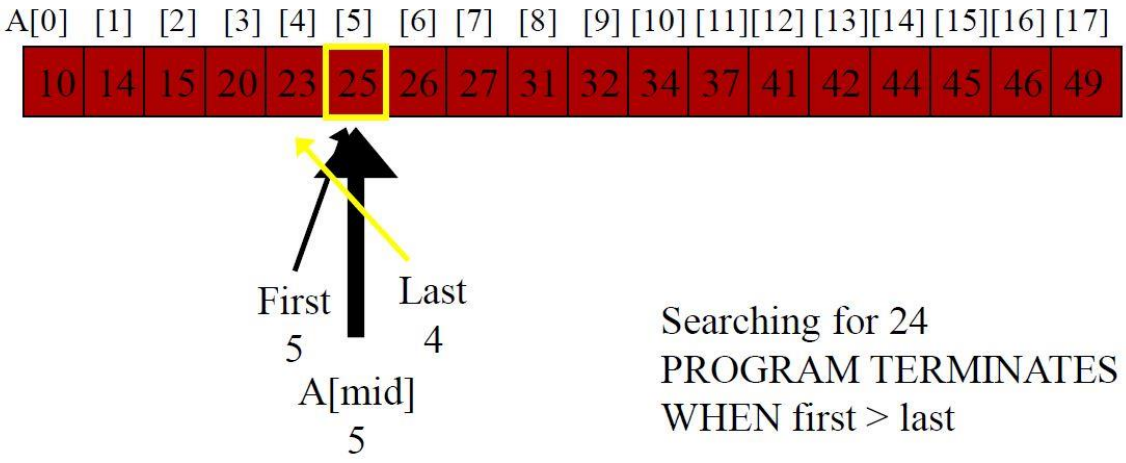
Binary search strategy



Binary search strategy



Binary search strategy



هنتكلم دلوقتي على طريقة سيرش اقدر بيها من مرة واحدة اعرف انا موجود فين في ال array بتاعي..هقدر اني ابص بس على ال key .. و اقولك بص يا معلم انت موجود في المكان الفلاني في ال array

الطريقة دي اسمها **Hash tables** و طريقة ال hashing دي مستخدمة في ال block chain and security:

Hashing

Hashing is the process of turning the key value into an integer pointer, used to locate a storage location in a larger array.

Hash codes should be designed to give different codes for different keys, although, this cannot be guaranteed.

Hashing

- Hashing is a technique that enables searching the data with a complexity **O(1)**.

- Using a key (index) value for each a data record, the key value can be used to represent an array index that is used for data insertion (no sorting time) and retrieval (no search time).

- The challenge in this design is to define a mapping function between the stored record data and the key/index value. Such a mapping function is called hash function.

ميزة كمان للطريقة دي ان الداتا وانت بتحطها في ال array مش لازم تبقى sorted

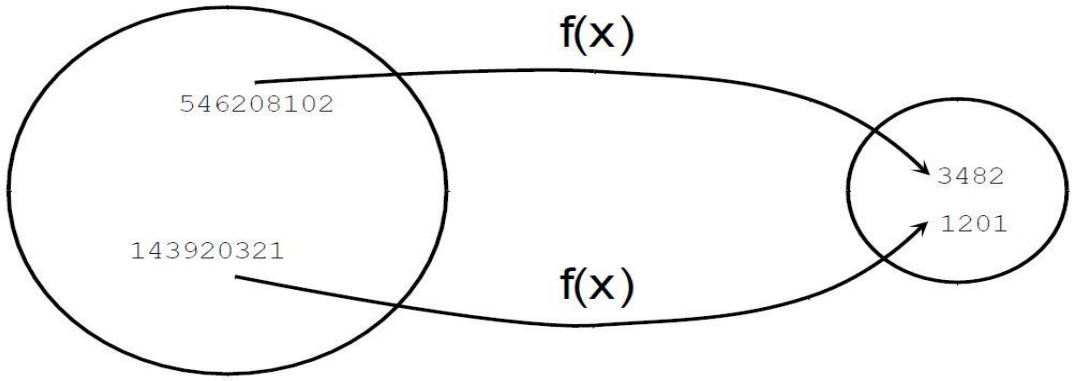
طب ايه التحدي يعني في الطريقة دي .. في الحقيقة ان الفكرة كلها انك دلوقتي عندك key اللي هو value معبرة عن الداتا somehow و عندي index جوه ال array .. و انا عايز اعمل mapping from the key to the index in the array by using hashing function or mapping function

وبالتالي ال mapping function دي لازم يكون ليها شوية properties و هي انها مثلا تكون

- One to one

لان مينفعش يبقى عندي اكتر من key و كلهم اعملهم mapping لنفس ال index in the array

Hash Function Map



Domain: 1 … 1,000,000,000 Range: 0 … 9999

Therefore, what is required is to design the hash function that takes in keys and gives out the indices of these keys in the array

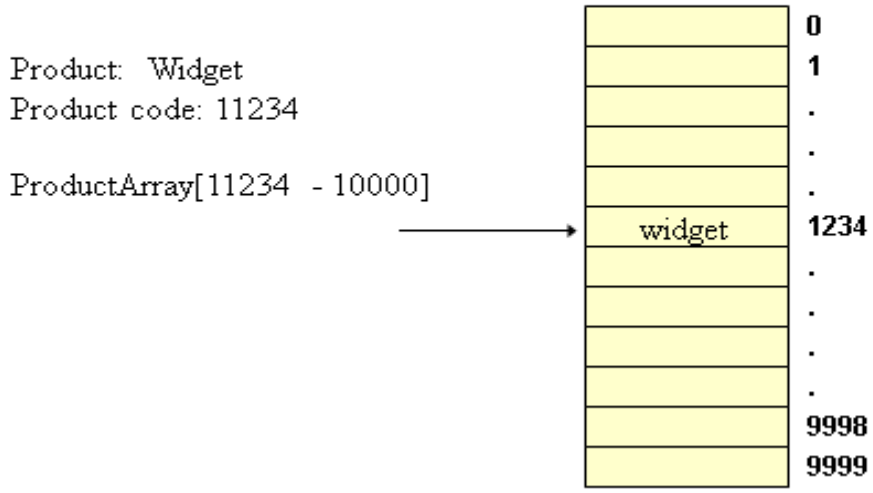
The same input in each time should give the same output

طب مثال عشان نفهم عشان حاسك اتلخبطت :

An example of hash conversion

- **You wish to store product information by product number. The product numbers have 5 digits with the lowest one being 10000.**
- **const int max_size = 10000;**
- **Then we could come up with a simple hash function**
- **hashKey = productCode – max_Size;**
- **This gives us a number between 0 and 9999.**
- **We can use this unique number to directly access the array element containing data for that product.**

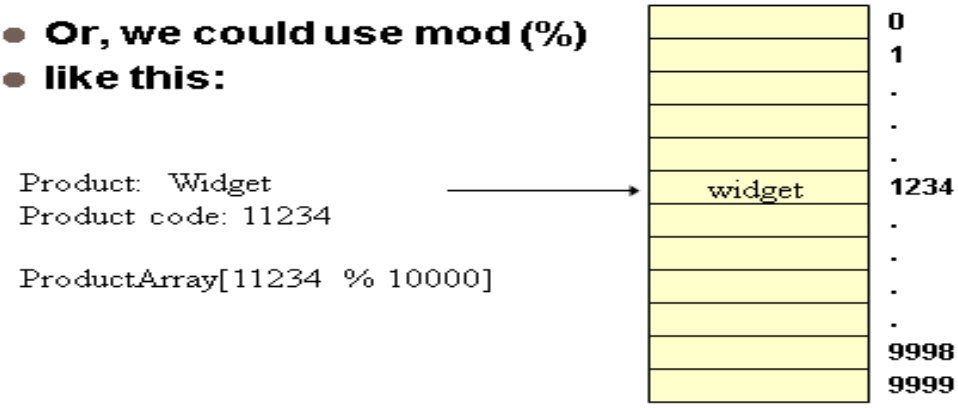
Product code hash example



دلوقتي كل product number هيجي هنطرح منه 10000 و يبقي ده المكان اللي المكان اللي هخذنه فيه في ال array لما اعوز بقي بعد كده ادور علي معلومات لل product number جه مش هفضل بقي ادور في 10000 مكان عشان الاقيه لا, انا علي طول هعمل التحويلة ديه اللي هي hash function و اروح علي المكان اللي هيطلعلي علي طول في ال array اجيبه في خطوة واحدة بس و عشان كده ده O(1) يعني ايا كان طوله ممكن توصل لل انت عايزة في خطوة واحدة بس .

Another implementation

- **Or, we could use mod (%)**
- **like this:**



و ديه طريقة تانية بدل ما اطرح 10000 باستخدام ال mod operator و هو مستخدم في كتيييييييييير جدا من ال hash functions . فهو طريقة مشهورة اوي .

و ده example لاستخدام ال hash functions في ال Networks

An internet example

- Internet Protocol (IP) uses 32-bit addresses to look up host names.
- Example: 63.100.1.17 (4 bytes)
- When you want to access a host machine that is not on your network the router takes the host name you give it and looks up the IP address.
- It then forwards your request to that host computer.

Problem

- Routers need to look up addresses as quickly as possible.
- There are millions of IP addresses, so how will it do this?
- Linear search? O(n)
- Binary search? $O(\log_2 n)$
- Neither of these are fast enough.

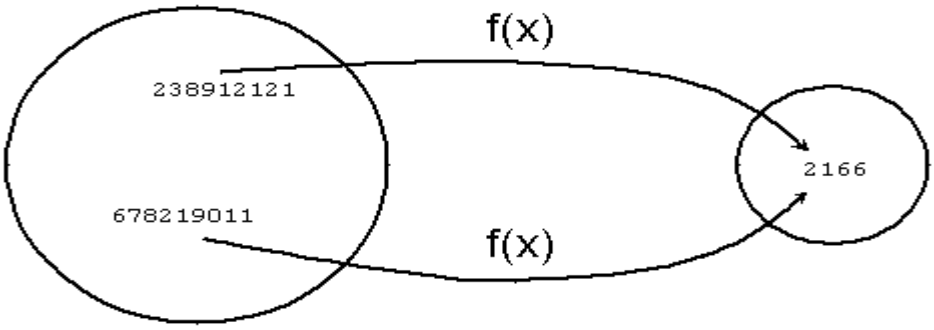
Answer

- **Convert the host name "www.cnn.com" to it's IP address using a hash function based on the characters.**
- **Perhaps we could use ASCII codes to generate unique numbers within a certain range.**

و طبعا باين اوي ان احسن و اقوي ميزة هي السرعة الرهيبة بتاعت ال hash functions عشان كده لا غني عنها في ال internet , يعني تخيل تبقي tedata بالسرعة ديه و يبقي كمان ال router شغال binary search ده يبقي نهار ازرق يا استاذ ممتاز XD

طيب , دلوقتي هنتكلم عن مشاكل و تحديات التي تواجه ال hash function technique

A Collision



-Collision: :

دلوقتي لو عندنا بقي system ضخم و فيه ملايين الارقام و الاماكن في احتمال كبير ان ال hash function ساعتها تتطلع نفس المكان ل اتنين keys مختلفين و ده اللي هو اسمه collision او اصدام يعني ال input يكون جي كده ولا علي باله و مبسوط و رايح ياخد مكانه من الهاش فنكشن بقي و اخر دلع و ياخد شنطه و يطلع علي المكان الي المفروض يتحط جواه هوب دبل كيك يلاقي في حاجة تانية متخذنه جوا في ماكنه :(

Issues in hashing

- **Each hash should generate a unique number. If two different items produce the same hash code we have a collision in the data structure. Then what?**
- **Two issues must be addressed**
  1. Hash functions must minimize collisions (there are strategies to do this)
  2. If (when) collisions do occur, we must know how to handle them.

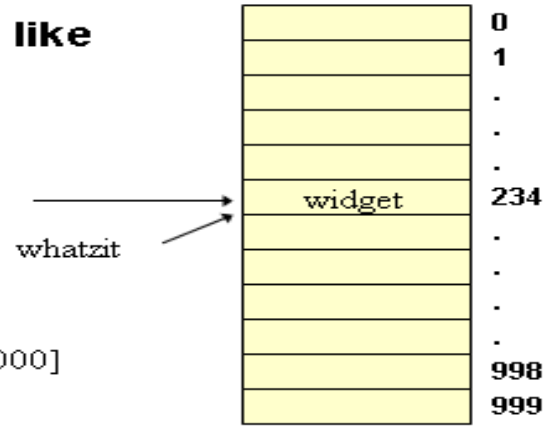
عشان ال سيستم بتاعي يكون محترم لازم اتعامل مع الحاجتين دول : اول حاجة اقلل ال collision ده علي قد ما اقدر و تاني حاجة اتصرف لو حصل عشان الداتا متضيعش.

What if...

- **We used mod (%) like**
- **this:**

Product: Widget
Product code: 11234
Product: Whatzit
Product code: 12234

ProductArray[11234 % 1000]



اللي فوق ده مثال علي ال collisionهنا المود للرقمين دول هيطلع نفس الindex

Two good rules to follow

- **A good hashing function must**
  - **1. Always produce the same address for a given key, and**
  - **2. For two different keys, the probability of producing the same address must be low.**

Collision

- Same hash value for two different records.
- Generally, a collision-free hash function is not easy to design.
  - A usual approach is to use an array size that is larger than needed. The extra array positions make the collisions less likely
  - A good hash function will distribute the keys uniformly throughout the locations of the array.

هنا بيقول انه صعب تعمل collision free hash function و احنا عايوين نقللها بس و حل للموضوع ده انك تزود طول ال array زايدة عن ما انت محتاج عشان تقلل احتمال ان يحصل collision يعني بس ده طبعا waste of memory.

عشان نحل بقي ال collision بعد ما يحصل :

Collison Resolution

- One way to resolve collisions is to place the colliding record in another location that is still open.
  - Open-addressing collision resolution techniques
    - Linear probing
    - double hashing
- **Chained hashing collision resolution techniques**

لو حصل collision في مكات كده هيبقي في واحد المفروض يتحط في المكان ده بس عشان هو مشغول فانا هحطه في حتة تانية بقي عشان ميضيعش مني :

اول طريقة هي ال Linear probing :

Linear Probing

- if hash(x) is full check for [hash(x) + 1].
  - A common problem with linear probing is ***clustering***
  - Clustering makes insertions take longer because the insert function must step all the way through a cluster to find a vacant location. Searches require more time for the same reason.

ديه سهلة و بسيطة جدا : بيقولك انت هات ال key و اعمل عليه ال operation و طلع ال index عادي خالص بعدين لو رحت هناك و لقيته مليان حد في اللي بعديه علي طول hash(x)+1 طب لو اللي بعديه مشغول بردو ؟ هنروح للي بعديه عادي جدا

hash(x)+2 لحد ما نلاقي مكان فاضي نحط فيه . الطريقة ديه فيها مشكلة انها بتعمل حاجة اسمها clusters او تكتلات يعني هتلاقي في اماكن معينة في ال array مليانة اوي و اماكن تانية فاضية .و ديه بتعطل العملية عشان افضل اعدي علي كذا واحدة مليانة لحد ما الاقي واحدة فاضية :(' .

عشان نحل مشكلة ال clustering ديه هنعمل الحل التاني :

Double Hashing

- Double Hashing is the most common technique to avoid clustering.
- Let *i = hash(key)*.
  - If the location data[i] already contains a record then let *i = (i+hash2(key))* and try the new *data[i]*.
  - If that location already contains a record, then let *i = (i+hash2(key))*, and try that *data[i]*, and so forth until a vacant position is found.

69

هنا بقي لو عملت اول hashing و لقيت المكان مشغول مش هحط في اللي بعده لا , هنا لو لقيت المكان مشغول هاخد ال key و ادخله علي hash function تانية و اخد ال o/p اللي طلع منها اجمعه علي المكان الاولاني و اروح احط فيه . الطريقة ديه تبعد الحجات عن بعض و مش هيبقوا متكتلين في مكان واحد .

اخر طريقة معانا النهاردة هي ال chained hashing :

Chained Hashing

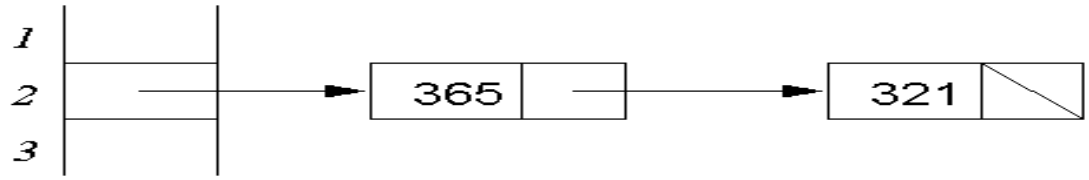
- In open addressing, each array element can hold just one entry. When the array is full, no more records can be added to the table.
- A better approach is called chained hashing
  - Each component of the hash table's array can hold more than one entry.
- We still hash the key of each entry, but upon collision, we simply place the new entry in its proper array component along with other entries that happened to hash to the same array index.
- The most common way to implement chaining is to have each array element be a linked list. The nodes in a particular linked list will each have a key that hashes to the same value.

70

ديه طريقة سهلة جدا و بسيطة : بيقولك لما يحصل collision و عايزين نحط اتنين في نفس المكان نعمل ايه ؟ نقولهم العبوا اخوات و نحطهم في نفس المكان , طب ازاي ؟ هنعمل array تانب parallel للاولاني و نسيبه فاضي و نفل نملا الاولاني لما تيجي حاجتين بقي عايزين نفس المكان نحط الحاجة التانية في نفس ال row بس في ال array التاني.

بس ديه طريقة تعبانة في طريقة احسن اني عمل array واحد و لما احتاج احط كذا حاجة في نفس المكان ازود cells في نفس المكان ب linked list و هي طبعا dynamic زي ما اخدنا فالطريقة ديه حلوة و efficient .

Chained Hashing



المحاضرة خلصت اخيرا ٨- ٨