CSC A48 – Unit 4 – Solving problems efficiently

1.- Are linked-lists efficient?

Up to this point, we have learned how to represent, store, organize, and search through a collection of possibly complex data items. We know what an ***Abstract Data Type*** is and why they are useful for coming up with solutions to problems in a way that makes the solution independent from a specific implementation. We studied lists, and their implementation as linked lists, and we used these for building a simple database able to answer a few queries on items in our collection.

You will no doubt use lists in the future for a variety of applications. So before we close our discussion on lists, it's worth taking time to think about how ***efficient*** they are as a solution to problems that require frequent look-up of items in the collection (whether it is to look at their content, update the data stored there, or carry out aggregate computations).

Suppose we wanted to use linked lists to implement a fully functional database engine. The database should be able to store information for a large number of data items (think big – tens of millions, hundreds of millions, or even billions of nodes in the linked list). The question is, ***what can we expect in terms of the time it takes for us to perform a search on the database?***

Having such a large number of data items in a collection is very common.

IBM's Watson used millions of documents to answer game questions for Jeopardy
Google Image Search was indexing well over 10 billion images by 2010
Facebook had about 1.5 billion daily active users in September 2018
Amazon sells over 3 billion different products worldwide!

So, thinking about ***what is the most efficient way to maintain a very large collection*** is of great importance.

To fully understand this question, we have to think a bit about ***how*** data was stored in the list. A common assumption, and one that is reasonable for many database applications, is that the data was added ***in no particular order***, or what is the same thing ***the entries in the list are randomly ordered*** with respect to any data fields we may want to use for answering database queries.

We also typically assume that ***queries come in in random order***. This is a reasonable assumption – think about Google searches arriving over a span of 1 minute from everywhere in the world – chances are, there will be no order or pattern to the searches carried out by people from Mexico, Australia, Singapore, and Finland, all of whom happen to be online at that particular time.

Finally, we have to provide some definition of ***efficiency*** so we can evaluate how well our list is doing. For a database engine, one reasonable measure is ***how many data items have to be inspected before we find the one we are looking for is found.***

(c) 2019 - Paco Estrada, Marzieh Ahmadzadeh

Now, recall from the last unit that to perform a search in a linked list, we have to do a ***list traversal***. This means starting at the head of the list, and traveling along the list's nodes until we find the one that contains the information we are looking for.

Question: How many nodes do we need to look at before we find the one we want?

- If we are ***super lucky***, the data we want will be in the ***head node*** and thus ***we only look at 1 node*** to find what we want, but with randomly ordered data, the chance of this happening is ***1/N***, where ***N is the number of nodes in our list***. For a list with 1,000,000,000 entries, the odds are very large against this ever happening!
- If we are ***super unlucky***, the data we want is in the ***tail*** of the list (or it is not in the list), and we have to look at ***all N nodes*** before we find it or can be sure it's not there.
- Most of the time, we are neither super lucky, not super un-lucky. The data is somewhere in the list, sometimes closer to the head, sometimes closer to the tail. Because the data is randomly ordered, the ***number of nodes we need to examine averages out to N/2*** after we run many, many queries.

This makes sense: Sometimes we find the data closer to the head, sometimes closer to the tail, but these two balance each other out so on average we have to look at about ***half of the linked list*** to answer any given query.

This is really not too bad if our list is a few thousand entries long. But once our linked list grows into the millions of items, and beyond, the ***cost of answering a query*** becomes too large. Simply put, our linked list ***has to look at too many data items in order to find a specific one***. This translates into a longer wait time for a program requesting information from the database. At some point, the list is so long that the wait time becomes impractically long.

2.- Why we need to look through all that data to find something

We may be inclined to think that the root cause of our problem is the structure of our linked list. And indeed, our linked list has a major limitation in that we can not access an element inside the list without doing list traversal. However, the root cause of our problem is not limited to the structure of the linked list.

Consider for example an ***array*** that contains the names of all of the hit songs from 2017 and 2018 (it's a small list so we can show an example here, but you should be thinking this applies to an array of any size).

I'm the One
Despacito
Look What You Made Me Do
Bodak Yellow
Rockstar
Perfect
Havana
God's Plan
Nice for What
This Is America
Psycho
Sad!
I Like It
In My Feelings
Girls Like You
Thank U, Next

The question we want to answer is:

Is searching for a specific item in an array ***with randomly ordered entries*** more ***efficient*** than searching in a linked list with the same (unordered) entries? Or, what amounts to the same thing, we are asking if using an array allows us to find what we want with less than ***N/2*** items examined on average.

Looking at the array above, it should be fairly clear that there is no pattern to how the data is ordered. If we need to find a specific song name, we have to start at the top and look through the array until we find what we want. ***This is called linear search***. Just like a linked list, there is a chance we get super lucky and our query is right at the first entry in the array, we can be super un-lucky so the query item is at the end, or not in the array; and on average we have to look through ***half the array*** before we find our information. If the array has ***N*** entries, we're back to looking through ***N/2*** entries on average.

This means that from the point of view of ***the number of items we need to look at to answer a query***, a linked list and an array are ***equally efficient***. While there may be a slight performance difference in terms of actual run-time (because there is more overhead to a linked list than an array), the small difference is not relevant as the collection size grows: both of these ways of storing information become inadequate for quickly answering queries.

3.- Organizing our data for efficient search

If we are to find a faster way to answer queries, we need to deal with the random order of our data. We need to sort it. Once our data is in sorted order, we can do a much more efficient job of searching for specific items.

Bodak Yellow
Despacito
Girls Like You
God's Plan
Havana
I Like It
I'm the One
In My Feelings
Look What You Made Me Do
Nice for What
Perfect
Psycho
Rockstar
Sad!
Thank U, Next
This Is America

For a person, looking up a specific song in the sorted array above is much easier. We understand the entries are in order, so we can easily ***find the spot where a particular song should be***. This is why all content indexes in the backs of books, library shelves, or the phone directory (back in the days when it was actually a printed book!) are sorted.

In programming terms, sorting data has the immediate benefit of making that data much easier to search, with the result that on a sorted array such as the one above, we only have to examine a few items in order to find what we're looking for.

Binary Search

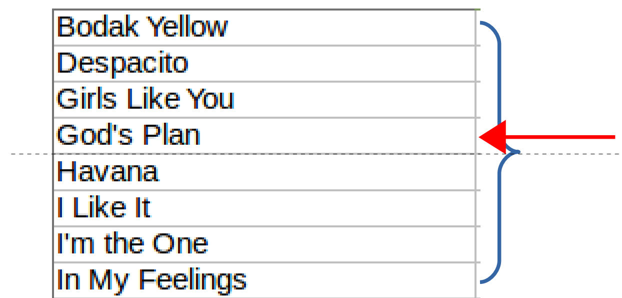
The process of finding an item in a ***sorted array*** is called binary search. Suppose we want to find the song "I'm the One" in the array above. The binary search process is illustrated below.

Bodak Yellow
Despacito
Girls Like You
God's Plan
Havana
I Like It
I'm the One
In My Feelings
Look What You Made Me Do
Nice for What
Perfect
Psycho
Rockstar
Sad!
Thank U, Next
This Is America

1) Consider the entire array, and find the item at the middle of the array (divide the length of the array by 2, and round down). ***Examine the item in the middle***.

* If **it's the item we're looking for** we're done
* Otherwise
    - If the item we're looking for is ***less than*** the item we just examined, it ***must be in the first half of the array***.
    - If the item we're looking for is ***greater than*** the item we just examined, it ***must be in the second half of the array***.

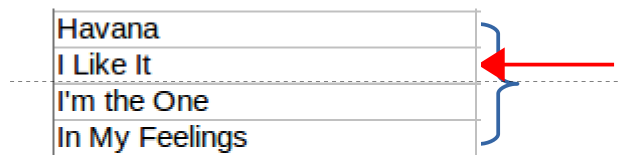
In the case of our array “I'm the One” comes before “In My Feelings” (remember, all entries are sorted alphabetically!), so we take the ***first half of the array.***



2) We perform exactly the same process on this chunk of the original array: Find the middle item and compare it to our query. Once more

* If **it's the item we're looking for** we're done
* Otherwise
    - If the item we're looking for is ***less than*** the item we just examined, it ***must be in the first half of the array***.
    - If the item we're looking for is ***greater than*** the item we just examined, it ***must be in the second half of the array***.

The song we want, “I'm the one” comes after the middle item “God's Plan”, so we need to take the second half of this array.



3) Repeat the same process until we find the item we want! “I'm the One” comes after “I Like It”, so we take the second half of the array above.

We found the song we were looking for! To do this, we had to examine ***4 entries***. In the un-sorted array

we would have expected to look at ***N/2 = 16/2 = 8*** entries to find a song (of course, allowing for being more or less lucky).

So it seems that sorting the array is doing something useful for us. Let's see if we can figure out what's happening:

* The binary search process uses the fact that the array is sorted to ***predict*** in ***which half of the array*** the data we want ***has to be***.
* This means that for ***every item we check***, we can ***discard half of the remaining entries***. When we choose which half of the array to search next, we can be sure we will never have to look at any items in the other half!
* So, for every step of binary search we complete, the number of items that remain to be checked is cut in half.

For example:

If our initial array has a length of ***1024*** entries:

* After the first step of binary search, we are left with ***512*** entries to check
* After the second step of binary search, we have ***256*** entries left
* After step 3, we have ***128*** entries left
* After step 4, we have ***64*** entries left
* After step 5, we have ***32*** entries left
* After step 6, we have ***16*** entries left
* After step 7, we have ***8*** entries left
* After step 8, we have ***4*** entries left
* After step 9, we have ***2*** entries left
* After step 10, we have ***1*** entries left

The number of items left to be checked is reduced ***very quickly!*** In the case above, by checking ***10 items*** we are able to find any item in an array of size ***1024***. And this is ***if we are very un-lucky*** and the item we want is the very last one we check!

Compare that with the expected ***N/2 = 1024/2 = 512*** items we have to check for ***linear search on*** an un-sorted array or a linked list, and it should be clear that binary search is a ***much more efficient*** method for finding information.

How many items does binary search have to examine in general?

From the above, we can see that at each step the number of items left to examine is cut in half. The question is, given an ***initial value for N, the number of entries in the array, how many steps are needed to reach the point where only one element is left?*** - this is equivalent to finding the ***maximum number of entries we need to examine to find what we want.***

The answer turns out to be $k=\log_2(N)$

For the array with ***1024*** entries, ***k=log₂(N)=10***, which is exactly the number of steps we had to do above to get to a single item. Let's see what this means in terms of how many items we need to examine as the number of entries in the array grows for three cases:

a) The ***maximum*** (worst case) number of items we need to look at for binary search
b) The ***average*** number of items we need to look at for linear search
c) The ***maximum*** (worst case) number of items we need to look at for linear search

N	***Binary Search: $log_2(N)$***	***Linear Search (avg): N/2***	***Linear Search (worst): N***
2	***1***	***1***	***2***
4	***2***	***2***	***4***
8	***3***	***4***	***8***
16	***4***	***8***	***16***
.			
.			
1,024	***10***	***512***	***1,024***
2,048	***11***	***1,024***	***2,048***
4,096	***12***	***2,048***	***4,096***
.			
.			
1,048,576	***20***	***524,288***	***1,048,576***
.			
.			
33,554,432	***25***	***16,777,216***	***33,554,432***

From the table above, it should be very clear to you that ***binary search is incredibly efficient***. Even in the worst possible case, with only 25 items examined we can find ***any one item in an array with over 33 million entries!*** Conversely, linear search on an un-sorted array or linked list would be expected to have to go through over 16 million items, on average, and all 33 million it we're unlucky before we find what we want.

3.- Computational Complexity

The ideas above can be made into a concrete, general framework for comparing different algorithms in terms of the ***computational cost*** of carrying out a task.

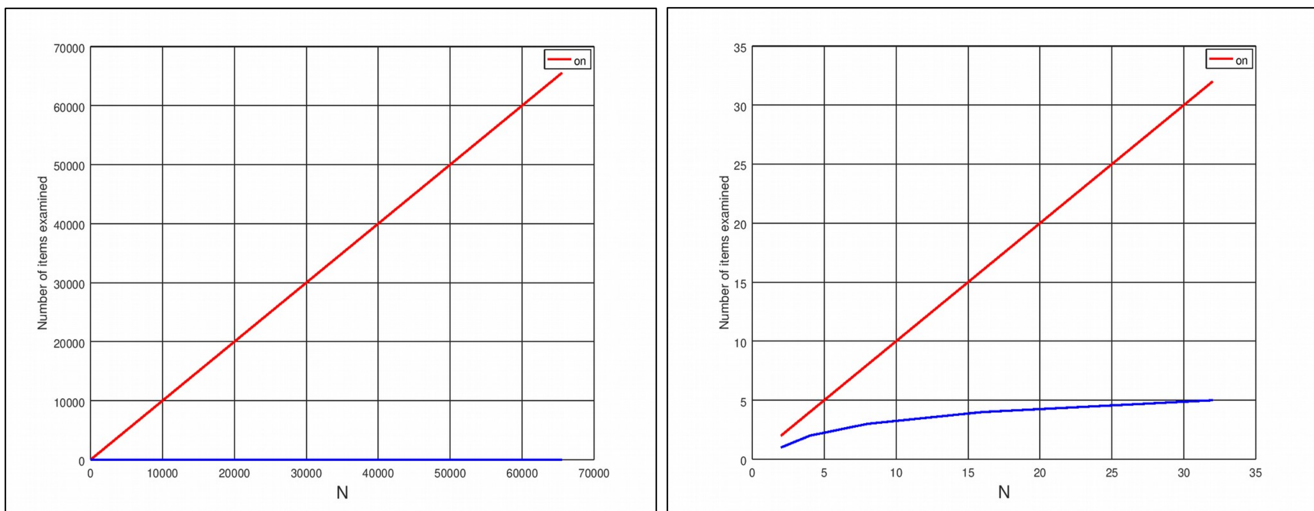
One key idea in understanding how we can compare different algorithms is that ***we want to find a measure of the amount of work a particular algorithm has to do as a function of N, the number of data items the algorithm is working on.*** We call this measure ***the computational complexity*** of the algorithm.

In the examples above, we saw that ***linear search*** has to examine ***N*** items in the worst case, while ***binary search*** only has to look at ***$log_2(N)$***. The reason for wanting to look at a function of ***N*** is

that we can ***predict*** how an algorithm will perform for increasingly larger data collections. In our search example, we have two different functions:

$$f_{\text{binary search}} = Log_2(N)$$
$$f_{\text{linear search}} = N$$

So we say that binary search has a complexity of ***$log_2(N)$*** and that linear search has a complexity of ***N***. Because the value of ***$log_2(N)$*** grows much more slowly than ***N***, we can conclude ***without having to test this on every possible array*** that ***binary search is much more efficient than linear search*** for large collections. We can visualize this by plotting both functions for increasing values of ***N*** and comparing the amount of work they have to do.



Drawing 1: Visualization of the complexity of two different search algorithms: Linear search (red) and binary search (blue). Binary search is incredibly efficient, to the point of looking like it does no work at all compared to linear search on large arrays! The shape of each function's curve is easier to compare on a graph for smaller values of N (right side)

It should be obious from the plots above which of the two algorithms is ***more efficient***. The observation we can make from this is:

> Given two algorithm and the functions of ***N*** that describe their computational complexity, the function that grows more slowly ***will always win-out as the number of data items grows***. So the algorithm with the slower-growing function is said to have a ***lower complexity***, and is therefore shown to be ***more efficient*** for large data collections.

Question: What can we say at this point about the ***run-time*** of two programs that do search on the same data, but where one uses binary search, and the other uses linear search?

What to take from the discussion above:

Being able to compare different possible ways of carrying out some task is essential for implementing good solutions to any problem. In computer science, the way to compare algorithms is by understanding ***its computational complexity***. Computational complexity is expressed as a ***function of N***, the number of data items the program has to work on.

The Big O Notation

In order to be able to better understand and compare the complexity of different algorithms, we use a special notation called the ***Big O*** notation. The importance of the ***Big O*** notation is that it allows us to think about the efficiency of algorithms in terms of general classes of functions. Here's what we mean by that:

Suppose that we have different implementations of ***linear search*** on arrays (perhaps written by different developers, in different programming languages), and the same for ***binary search***. We then set out to carefully measure their ***run-time*** for arrays of different sizes, and we find that they behave as follows:

a) .75*N (e.g. for N=10, this implementation runs in 7.5 seconds)
b) 1.25*N
c) 2.43*N
d) 1.15 * Log_2(N)
e) 1.75 * Log_2(N)
c) 15.245 * Log_2(N)

The results above require some thought – ***we know*** how much work *linear search* has to do to find an item in the array, and that in the worst case that would be looking at all ***N*** entries in the array. But we hadn't considered what could happen when variations between implementations are introduced.

Perhaps an implementation in C will be slightly faster than one in Java, and perhaps the one in Java will be somewhat faster than one written in Matlab. But the important thing to note is that ***all of them are linear functions of N***. It is a property of ***the linear search algorithm.*** The only thing that can change among (correct) implementations is the constant factor. We can say exactly the same in a different way: Any correct implementation of the ***linear search*** algorithm will have a complexity that is characterized by ***c*N***, where ***c*** is some constant, for large values of ***N***.

The same holds true for different implementations of ***binary search***. They all are ***logarithmic functions of N*** multiplied by some constant that is implementation dependent.

We want a way to compare algorithms in an ***implementation independent*** way. We need to know ***which is the better algorithm, the one requiring less work*** to solve a given task. And if we can analyze algorithms in this way, we can always pick the most efficient one. The Big O notation makes explicit the ***fastest-growing function of N*** that characterizes the complexity of some algorithm. The actual definition states that

$$f(x) = O(g(x))$$

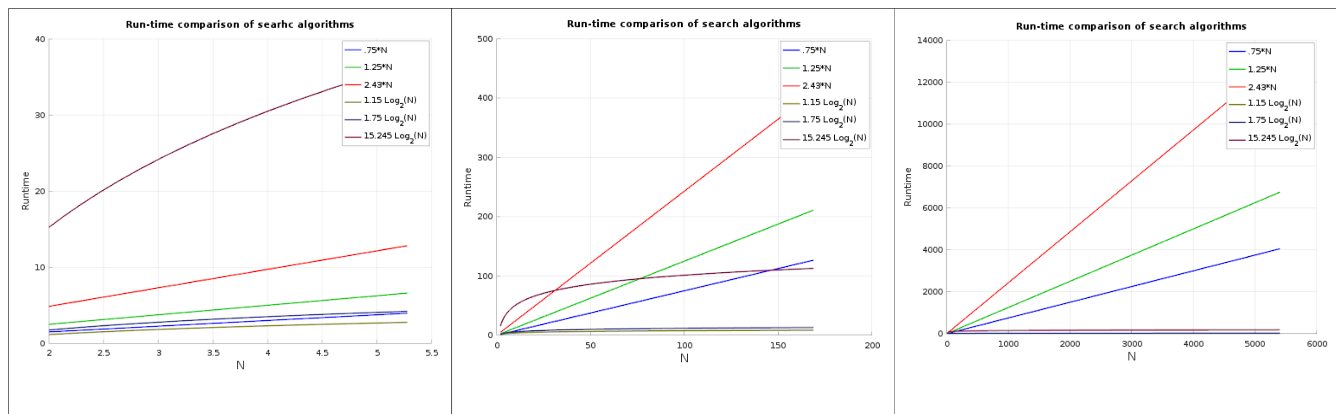
if and only if, there exists some positive constant ***c*** such that for sufficiently large values of ***x***,

$$|f(x)| \leq c \cdot g(x), \quad x > x_0$$

Put in a different way, to say that the ***run-time f(x) of some algorithm is of order O(g(x))*** means that the run-time is less than, or equal to ***c*g(x)*** for some constant *c* assuming ***x*** is large enough. In the case of our search algorithms above, we can state that:

- Linear search has a complexity of ***O(N)* –** read as “the complexity of linear search is of order N”
- Binary search has complexity of ***O(Log₂(N))*** – read as “the complexity of binary search is of order Log₂(n)”

In fact, we can go a bit further and say that the complexity of binary search is ***O(Log(N))***, without needing to worry about the fact that the logarithm is base 2. ***Why is this reasonable?*** Consider the graphs below:



Plots of run-time for the search algorithms described above. For small values of *N*, *linear search* would appear to be more efficient. However, by the time *N=150* all *linear search* implementations have overtaken the *slowest binary search* implementation, and by the time *N=1000* it's all over for *linear search*. The conclusion is: *Binary search* will win out in the end, regardless of how it's implemented, if *N* is large enough.

The point not to be missed in the plots above is that ***the algorithm whose Big O complexity is slower-growing will win in the end, for large enough N***. The *Log(N)* function (with any base) is *much slower growing* than *any linear function*, so we *will always expect binary search to be faster on a large enough array*.

From algorithm complexity to problem complexity

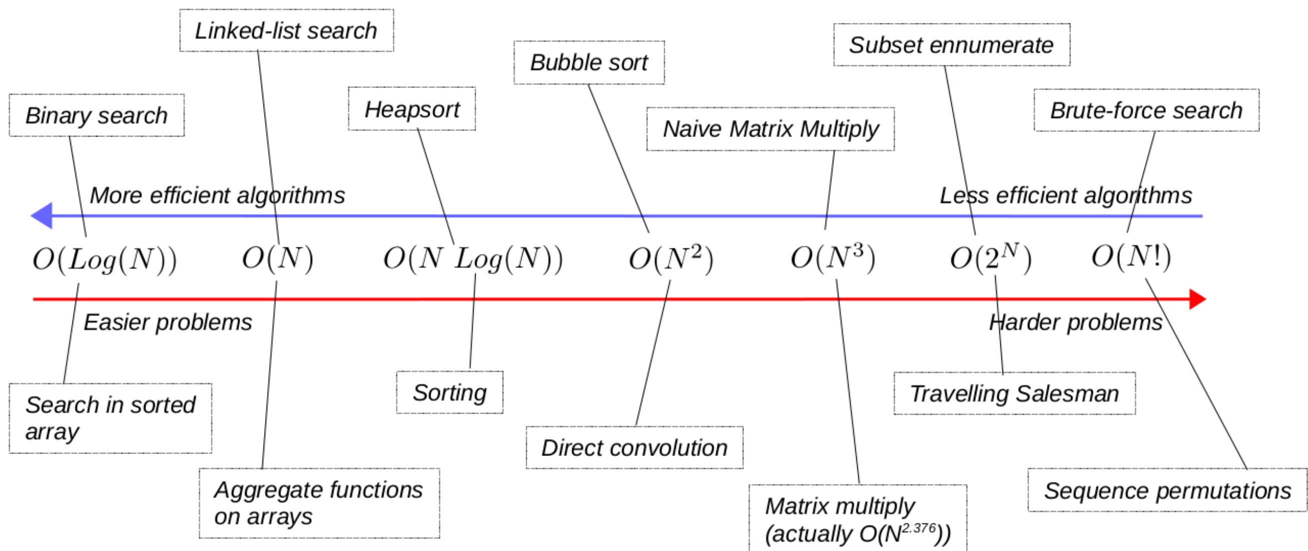
We started with the goal of understanding how efficient two different algorithms for finding a particular item in an array can be. Complexity analysis is a wonderful tool for doing this. However, there is an even more powerful idea behind the use of complexity analysis in computer science:

We can study, quantify, and prove results regarding ***the complexity of a given problem***. What is the difference? Let's take an example: Sorting an array (and we will come back to this example soon in more detail!)

- ***Algorithm complexity*** means we can look at different algorithms to sort an array, and figure out which one is going to be more efficient as the array size grows.
- ***Problem complexity*** means we study the actual problem of sorting, and we try to figure out ***what is the theoretical lower-limit on how much work the best possible algorithm has to do*** to sort an array.

For example, proving that ***linear search*** has a complexity of ***O(N)*** is equivalent to showing that it is not possible to find an algorithm that can do ***linear search*** with complexity less than ***O(N)***.

This is a powerful and important idea because it allows us to ***classify*** problems by ***how hard*** they are to solve computationally. Harder problems are characterized by a ***Big O*** complexity given by fast-growing, or very-fast growing functions. As a developer, it's important to become familiar with the *different classes of problems* you may have to work with one day, and to consider carefully what is reasonable to expect of an algorithm, given the complexity of the problem it is working on.



The plot above shows examples of algorithms (on top) and problems (at the bottom) with different ***Big O*** complexity. Note that for given problem (e.g. sorting), you can easily find algorithms whose complexity is greater (bubblesort) than the best known complexity bound for the problem. Knowing what the best known complexity estimate is for a given problem allows you to evaluate how good an algorithm is that solves *that* problem.

Next year, in B63 you will learn to analyze algorithms to figure out what their complexity is, and you will learn about different ways to measure, quantify, and express what we know about the complexity of different algorithms and operations on different ADTs.

For now, don't forget that we can measure complexity in terms of different quantities: *the number of times an item in a collection is accessed, the run-time of an algorithm, the number of mathematical operations a certain function has to perform*. Which measure to use depends on what problem you're solving, but the analysis, and what it tells you about an algorithm or problem in terms of how much work is being done to solve it, is understood in the same way.

Also, don't forget that ***computational complexity analysis*** applies to every type of problem, not only searching for information in a collection. It is incredibly important in all areas of computer science that deal with numerical computation, including machine learning and artificial intelligence. Make sure you build a sound understanding of the ideas above, and next year expand and strengthen it in B63.

So this means the implementation doesn't matter right?

Not quite – solving a problem efficiently requires you to ***first think of the complexity of your algorithm for solving the problem***. But once you have satisfied yourself you have an algorithm with the best possible complexity, you need to write a good implementation for it! Once we have selected an algorithm with the *right complexity* for a problem, we are at the point where we *now care about those pesky constant factors!*

Going back to our original example of search algorithms, the difference between the implementations becomes important once ***N*** is large enough, e.g. ***N=1000000:***

$1.15 * \text{Log}_2(N) = 22.9$
$15.245 * \text{Log}_2(N) = 303.6$

So the implementation makes all the difference between you having to wait around 20 sec. for your result, or having to wait over 5 minutes!

Exercise: Consider a function that takes 2 input arrays:

```
void multiply_accumulate(float input[N], float output[N])
```

The function computes each value in the '*output'* array as

$$output[i] = \sum_j input[i] * input[j]$$

That is, the ith entry in the *output* array is the result of multiplying the ith entry in the *input* array with every other entry in the *input* array (including itself), and adding all of these up.

One way to implement this function is shown below:

```
void multiply_accumulate(float input[N], float output[N])
{
     int i,j;

     for (i=0; i<N; i++)
     {
          output[i]=0;
          for (j=0; j<N; j++)
          {
               output[i]=output[i]+(input[i]*input[j]);
          }
     }
}
```

Question: What is the ***Big O*** complexity of the function shown above? (for this function, define the complexity in terms of how many times entries in the *input* array are accessed) If you're having a tough time figuring that out from the code, try and list the operations the loops are doing for a small value of ***N*** and see if that suggests something to you.

Question: Would the ***Big O*** complexity change if instead of *accesses to entries in the input array* we defined complexity in terms of *the number of multiplications* that are carried out by the function? (this definition is more likely to be tied to the run-time for this function, since it requires not only accessing information, but also performing computations on data)

Question: Is this the best we can do for this problem? Is there a better algorithm? And if there is, what is the ***Big O*** complexity of that implementation?

4.- Back to our problem – How to make searching for information more efficient

We started this unit trying to understand how efficient linked lists are in terms of letting us search for items in our collection. We have now seen that searching in linked-lists has a complexity of ***O(N)***, and because of that, for large collections, they are not the optimal way of storing information that will need to be searched frequently.

We also saw that ***binary search*** has a complexity of ***O(Log(N))*** which makes it incredibly efficient for finding information even for very large collections. So based on this we may conclude that we should give up on linked-lists and that we should simply stick to arrays of sorted items so we can have efficient search.

However, we left arrays behind when we realized that their limitations (fixed capacity, either insufficient for data, or wasteful of space) made them hardly a good solution for the management and storage of large collections of items whose quantity is not known in advance, and in cases where the size of the collection can change a lot over time.

So we have a serious problem now:

At this point it looks like we can have either

- A dynamic data structure that allows us to organize and maintain a large collection of items without wasting space.

or

- An array, with fixed capacity and its associated problems, but where we can perform binary search to efficiently search for information.

What are we missing?

Suppose we decided that efficient search is more important to us than not wasting space, so we're willing to use an array, large enough for our collection, so we can do binary search and find items quickly. Sounds like a good plan, *except that we haven't accounted for the fact that we expect items to be added to our collection in arbitrary order*. Binary search requires our array to be sorted.

So before we decide out best bet is to use a sorted array along with binary search, we have to consider the ***cost*** (and you know now that by this we mean the ***computational complexity***) of sorting the array in the first place.

The cost of sorting

Consider an (unsorted) array with ***N*** entries. Let's cosider the simplest sorting algorithm we can come up with: ***bubble sort***. If you haven't seen bubble sort before, here's a simple implementation:

```
void BubbleSort(int array[], int N)
{
   // Traverse an array swapping any entries such that
   // array[j] > array[j+1]. Keep doing that until the
   // array is sorted (at most, N iterations)

   int t;

   for (int i=0; i<N; i++)
   {
     for (int j=0; j<N-1; j++)
     {
       if (array[j]>array[j+1])
       {
         t=array[j];
         array[j]=array[j+1];
         array[j+1]=t;
       }
     }
   }
}
```

The function proceeds through the array, repeatedly swapping elements so that the largest one

moves toward the end of the array. At the end of each iteration, one more entry has been definitely placed in its right place (in fact, if you think about it carefully, the loop for *j* above can be shortened to `for (int j=0; j<N-i-1; j++)` since the last *i* entries in the array are sorted after the ith iteration).

Try it out on an unsorted array. The function will produce an array sorted in ascending order, as required for binary search, but, at what computational cost?

Consider the nested loop structure of the function if we are counting array accesses:

```
Outer loop has N iterations
     Inner loop has N-1 iterations
          The inner loop updates at most 2 array entries
```

So the complexity of the function is $N(N-1)2 = 2(N^2 – N) \rightarrow O(N^2)$. The complexity of bubble sort is of order N^2 (notice we ignore the N term – why do we do that?).

This complexity result holds even if we use the shorter form of the loop on *j*, which has only *N-i-1* iterations. The conclusion is:

If we want to use an array to find items with complexity ***O(Log(N))***
First we have to sort the array, which has complexity $O(N^2)$

Suddenly using a sorted array looks much less appealing – the quadratic cost of sorting the array would completely swamp the advantage gained by using binary search.

Caveats:

- If our data is added to the collection at one time (and no later insertions or deletions are expected) then we can argue that the cost of initially sorting the array is *divided among all the search operations* that will be carried out later. For example, if *N=1000*, after one million queries, the added cost of sorting comes down to ***one*** additional data access per search!

- Conversely, if data is constantly being inserted/deleted, the cost of sorting dominates over the savings obtained from binary search. In this case, we would prefer a linked-list.

Wait a minute! President Obama said not to use bubble sort!

Illustration 1: "I think the bubble sort would be the wrong way to go."
(Photo: U.S. Federal Government – public domain)

Bubble sort is definitely ***not*** the best sorting algorithm out there – we sometimes use it for small arrays because it's so simple and quick to implement, but we can do much better. If you recall from our discussion on the ***computational complexity of problems*** above, the best known sorting algorithms have a complexity of ***O(N Log(N))***. There are several algorithms that can reach this level of performance, including a couple you will study in detail in B63 next year. For now let's see what we get if we replace our costly bubble sort with a more efficient (on average) ***quick sort***.

Quick sort works by choosing one entry in the array (called a *pivot*) and then using this pivot to split the array into elements that are *less than the pivot* and those that are *greater or equal* to the pivot. Then the process is repeated on each sub array. The process ends when all sub-arrays have zero or a single element (at which point the data is sorted). The sorted array is constructed from the sorted sub arrays.

There are many optimizations, and the process can be carried out without using additional array space for the sub arrays (it can be done *in place*). There are also many ways to choose the *pivot*. An analysis of quick sort is beyond the scope of this course (you'll likely see it in detail in B63), but the important results are that

- The average-case complexity of quick sort is ***O(N Log(N))***.
- The worst-case complexity of quick sort is ***O(N^2)***.

The difference in the two statements above is very important – given a randomly ordered array, we can expect the cost of doing quick sort to be close to the average case complexity of ***O(N Log(N))*** which is good. However, if we are very unlucky (when we picked for *pivot* the *smallest* or the *largest* entry in a sub array) all the entries other than the pivot end up in one of the two sub arrays – if this happens at each step of the process, we are looking at the worst case complexity ***O(N^2)***, same as bubble sort!

We have to be quite unlucky to hit the worst-case, so you will find that quick sort is very often used in practice. Let's assume that we use quick sort to keep our array sorted and then rely on binary

search to find information. Have we solved our problem?

Our search engine now has to do the following work:

- Sorting the collection, which with a bit of luck has complexity ***O(N Log(N))***
- Searching the collection for items, using binary search, with complexity ***O(Log(N))***

Like before, if we assume the data is added to the collection at one time, and no insertions or deletions happen later, we can split the cost of sorting among all the searches we run on the collection. For *N=1000*, now it is enough to run about 5,000 searches to get to the point where the additional cost of sorting comes down to ***one*** extra item access per search (compared to 1,000,000 searches we required if using bubble sort).

Clearly this is much better than using bubble sort, and starts to look like a realistic solution. However, in practice we expect to have our data change, there will be insertions, deletions, and modifications to existing items. If we sort the array each time this happens, once more the cost of sorting dominates the total computational cost of search, and we're back to looking at a regular linked list as the better solution.

So have we found an efficient way to make a large collection searchable?

Not quite...

- The array solution has space limitations (fixed size can be either insufficient, or wasteful of space)
- It requires sorting. With the best sorting algorithm this has complexity ***O(N Log(N))*** which is already worse than ***linear search***.
- We expect our collection to change over time – insertions, deletions, and modifications will require work to ensure the array remains sorted.
- The conclusion is that ***sorted arrays + binary search*** is not necessarily the best solution for organizing, storing, and searching over a large and changing collection of items.

What do we need?

- It is clear that for efficient search we need sorted data
- We can not keep an array sorted in an efficient manner
- And we want the ability of linked-lists to request and use space on-demand

So what we want is ***a data structure*** that allows us to:

- Request space on-demand, with no fixed limitations on how many items are in the collection
- Keeps our data sorted at a ***lower computational cost*** than that of re-running a sorting algorithm every time.
- Allows us to search for data efficiently (ideally, as efficiently as with ***binary search***).

We will now see one such data structure, study its properties, and discuss some of its applications.

5.- Trees, Binary Trees, and Binary Search Trees

Recall that a linked-list is a data structure in which nodes contain one item from a collection, and one link to a successor node which is the next node in the list.

A tree is a generalization of this idea, it consists of nodes, each of which contains one or more data items from a collection, and ***one of more links*** to ***children nodes*** (the equivalent of the successor node, but now we can have many!).

Trees are extremely common structures in computer science, used for a wide range of purposes.

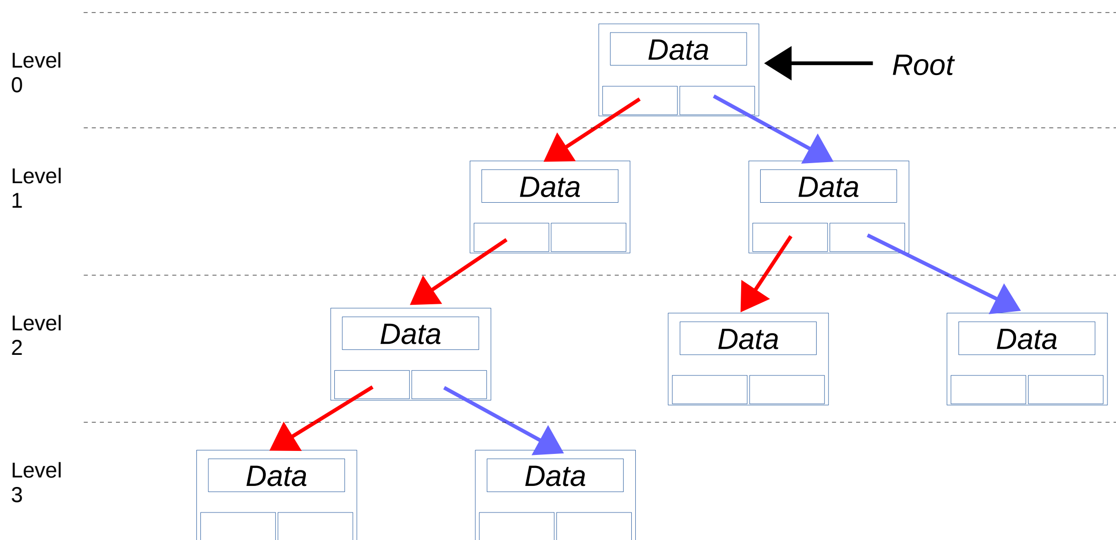
Illustration 2: Computer graphics uses tree structures to create, manipulate, and render virtual trees, plants, and articulated models such as animals. Photo: Solkoll, Wikimedia Commons, public domain

A particularly common type of tree is the ***binary tree***, which has the property that each node has ***two children nodes***, the ***left child***, and the ***right child***.

The diagram below illustrates a binary tree. Note the following:

- Each node has two spaces for links – one for the left child (shown with red arrows), and one for the right child (shown with blue arrows)
- Nodes may have zero, one, or two children

- The node at the top of the tree is called the ***root node.*** Similarly to the ***head of a linked list***, we manage a tree by keeping a pointer to the root node.
- Each ***level*** in the tree consists of nodes that are at the same ***distance*** or ***depth*** with regard to the root node. Each level contains up to 2 times the number of nodes of the previous level
- ***Leaf nodes*** are nodes ***with no children***.



Binary Search Trees

A Binary Search Tree (BST) is binary tree such that for each node in the tree, the BST property holds:

- ***Data*** in the nodes on the ***left sub-tree*** of a node have value ***less than, or equal to*** the value of the data in the node.
- ***Data*** in the nodes on the ***right sub-tree*** of a node have value ***greater than*** the value of the data in the node.

This means that at each node, we can quickly determine whether:

- The data is in the current node
- The data is not in the current node, but if it is in the tree, it must be in the left sub-tree
  - or -
- The data is not in the current node, but if it is in the tree, it must be in the right sub-tree

At this point, you should be thinking about binary search! The ***BST*** is intended to allow us to organize a large collection in such a way that we can quickly search through it. Indeed, under the assumption that our data is inserted into the tree in random order, the ***BST*** can provide search complexity of ***O(Log(N))***. Let's consider the search process in a ***BST***:

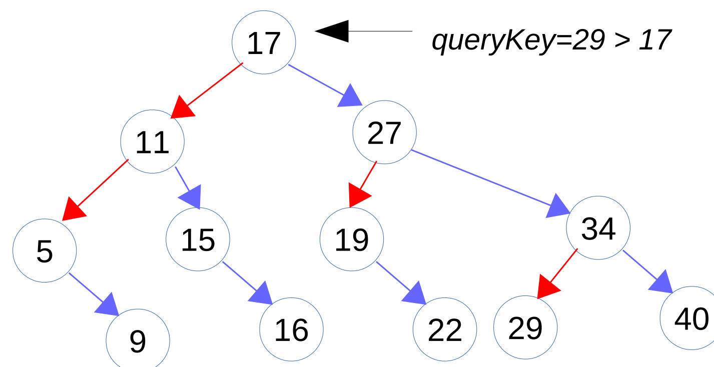
searchForKey ← subtree, queryKey

if the queryKey is equal to the key at the root node of the subtree
Found the key! Return a reference to the root node of the subtree
else
if the queryKey is ***less than or equal to*** *the key in the root node of the subtree*
searchForKey ← left subtree, queryKey
otherwise
searchForKey ←right subtree, queryKey

The search process starts at the top of the tree, checking the query value against the value stored at a node. If the value is found the search succeeds and returns a reference to the node, otherwise, it checks whether the query value should be in the left or the right subtree, and continues searching for it along that subtree.

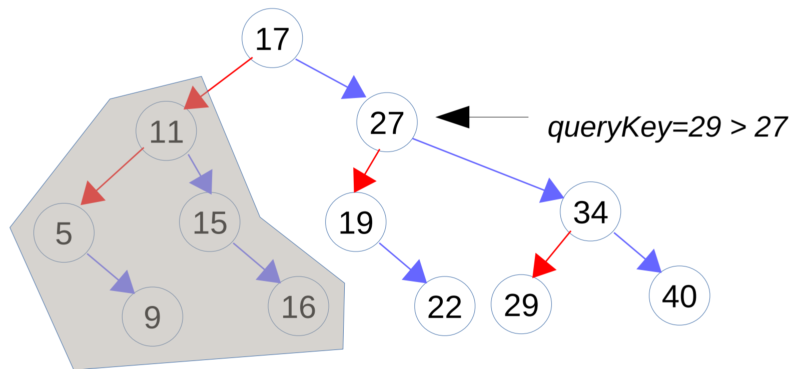
With BSTs, we normally call the values used to search for information in the tree ***keys***. If the tree contains data items which are simple data types then the ***keys*** and the ***data values*** are the same. But remember that we intend to use these data structures to organize and maintain large collections of ***compound data types***. So in general, the ***key*** will be ***a suitable field, or a subset of fields from the compund data type***. Keys have to be comparable (i.e. besides checking for equality, we must be able to tell which of the two keys is greater and which is lesser according to some ordering that makes sense for our data). This often involves writing a comparison function that works with our data type.

The search process is illustrated below

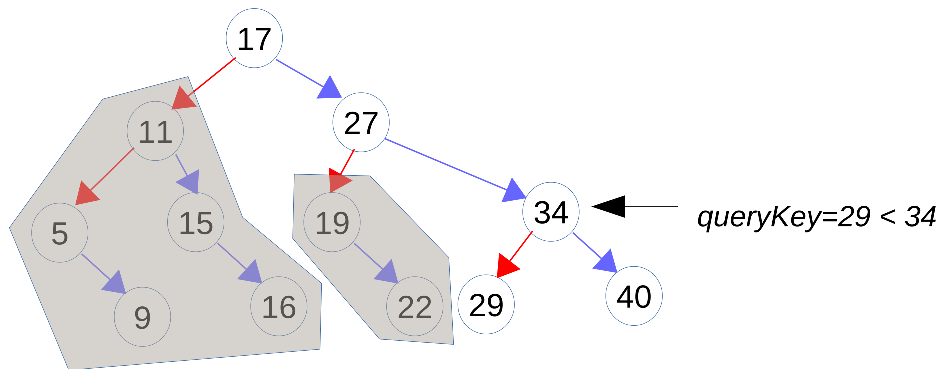


Starting with the root, the *queryKey* (29 in this case) is compared with the key in the node. In this case the *queryKey* is *greater than* the key in the node, so we know the *queryKey*, if it is in the tree, *must be in the right subtree.*

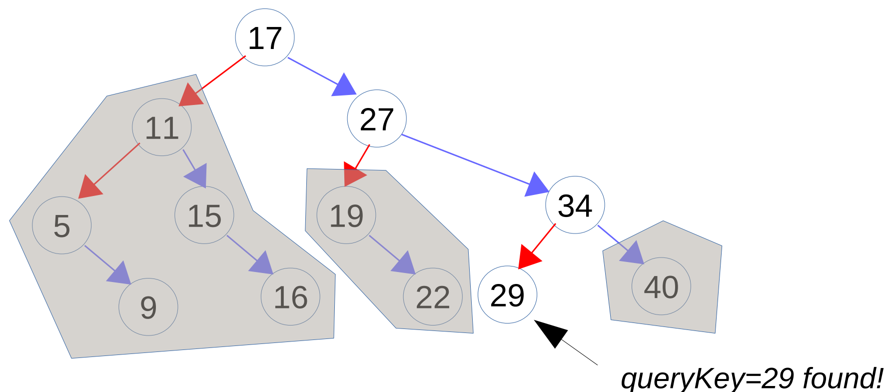
The search process now continues along the right subtree, which means ***about half the remaining keys are discarded from having to be searched!***



The *queryKey* is *greater than the key at the node (27)* so we know the value we want *must be in the right subtree*. The search continues along the right subtree *discarding from search roughly half the remaining keys*.



At this point, the *queryKey* is *less than the key at the node* so we know if it is in the tree it *must be in the left subtree* and continue our search there. Once more, discarding from search roughly half the remaining keys.



And we have found our *queryKey!* Notice in the final diagram *how many keys in the tree were discarded*, meaning the search process would *never have had to check them during the search*.

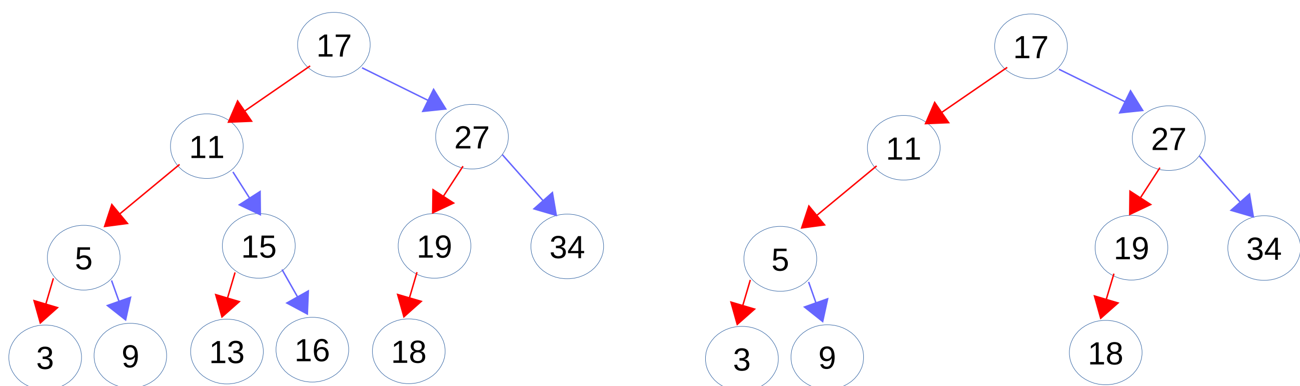
You can see how, similarly to *binary search*, the search process in a ***BST*** quickly discards from

search a large portion of all the values stored in the tree. The question is ***how many items need to be examined for us to find the query key (or determine it's not in the tree)?***.

The search moves down one level in the ***BST*** for every comparison between the *queryKey* and values in the tree. This means that *at most*, the search has to examine a number of nodes equal to *the depth of the **BST*** – the number of levels in the tree.

So, ***how many levels are there in a full BST?***

A ***full BST*** is a ***BST*** in which each level except for the last one is full (all the available nodes have data, no un-used nodes). The last level may or may not be full, but if it is not full, then the keys are packed toward the left of the tree.



Drawing 2: Left: A full BST, levels 0-2 are complete, the last level has all its keys packed to the left. Right: Not a full BST, level 2 is missing keys, nodes in the last level are not packed to the left.

In a ***full BST***, each level contains ***as many nodes as all the levels above + 1***. For instance, level 2 contains 4 nodes, whereas there are 3 nodes above level 2 in the tree. Level 4 contains 16 nodes, with levels 0-3 containing a total of 15. This means that for every additional level we add to the ***BST***, we *double its capacity for storing keys*.

Therefore, a ***BST*** with ***k levels*** can store up to $N=2^k$ keys, and therefore, for a collection with ***N*** keys, we need a ***BST*** with at least $Log_2(N)$ levels.

The height of a ***full BST*** storing ***N*** keys is $\lceil Log_2(N+1) \rceil$ (the funky brackets represent the ceiling function). Searching for a key in such a tree will therefore have a complexity of ***O(Log(N))***. Same as binary search!

Unfortunately the ***BST*** will in general ***not be full***, and so the ***O(Log(N))*** search complexity value we obtained above is not the whole picture. To understand why that is the case, however, we have to implement the basic operations supported by ***BSTs***.

Operations on BSTs

The ***Binary Search Tree*** must support at the very least the following operations:

- Initializing an empty binary tree
- Inserting a node into the ***BST***
- Removing (deleting) a node in the ***BST***
- Searching for a specific item in the collection

Additional operations are sometimes defined, but they are often special cases or combinations of the operations above.

6.- Implementing a BST

Let's improve on the design of our *Kelp* app by replacing the linked list we were using to keep track of restaurant reviews by a ***BST*** (we hope that by doing this, review look-ups will be much faster). The definition of the *'Review'* composite data type that stores the actual reviews does not change, as a reminder:

```
typedef struct Restaurant_Score
{
     char restaurant_name[MAX_STRING_LENGTH];
     char restaurant_address[MAX_STRING_LENGTH];
     int score;
}Review;
```

What is different now is the definition of the ***nodes*** that will store the reviews in our ***BST***. As it turns out, the ***BST*** node is very much like a linked list node, but with two pointers instead of one. These correspond to the ***left*** and ***right*** children of the node:

```
typedef struct BST_Node_Struct
{
     Review rev;                        // Stores one review
     struct BST_Node_Struct *left;     // A pointer to its left child
     struct BST_Node_Struct *right;    // and a pointer to its right child
} BST_Node;
```

If you wanted to create a ***BST*** for diferent data types, you would only need to change the data component in the node definition above.

Same as with the linked lists, we need to write a function to *allocate and initialize a new 'BST_Node' on demand*.

```
BST_Node *new_BST_Node(void)
{
     BST_Node *new_review=NULL;        // Pointer to the new node

     new_review=(BST_Node *)calloc(1, sizeof(BST_Node));
```

```
    // Initialize the new node's content (same as with linked list)
    new_review->rev.score=-1;
    strcpy(new_review->rev.restaurant_name,"");
    strcpy(new_review->rev.restaurant_address,"");
    new_review->left=NULL;
    new_review->right=NULL;
    return new_review;
}
```

Other than the names, the only difference between the function above and the one we used with linked list nodes is highlighted by the red box: We have now two pointers that need to be set to *NULL*.

Inserting nodes into a BST

For linked lists, we had to keep around a pointer to the *head of the list*. In the case of a ***BST***, we will need to keep around a pointer to the ***root*** of the tree (the node at the top). The insert process ***must ensure that the BST property is enforced when a new key is inserted in the tree***.

This means that keys in the tree are inserted so that for any node in the tree, keys in the left subtree are smaller than or equal to the key in the node, and keys in the right subtree are greater than the key in the node. Since the data in our ***BST*** is a composite type *'Review'*, we need to define which field we will use as ***key***. Let's make our ***BST*** keep our reviews ordered by the name of the restaurant. So in the implementation of the ***BST*** shown below, whenever we talk about the ***key*** for a given node, we are in effect talking about the *'restaurant_name'* field of the '*Review'* component of the ***BST*** node.

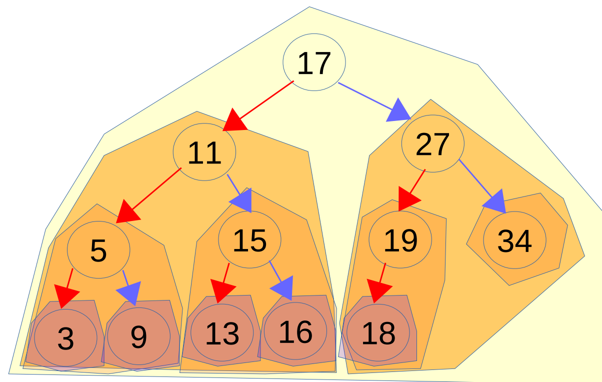
Let's look at how the insert function is defined, and see examples of how it works. It takes as input two parameters: A pointer to the ***root*** of a ***subtree*** where we are inserting a new node, and a pointer to the *new review* being inserted in the tree

```
BST_Node *BST_insert(BST_Node *root, BST_Node *new_review)
{
    if (root==NULL)               // Tree is empty, new node becomes
        return new_review;        // the root

    // Determine which subtree the new key should be in
    if (strcmp(new_review->rev.restaurant_name,\
                root->rev.restaurant_name)<=0)
    {
        root->left=BST_insert(root->left,new_review);
    }
    else
    {
        root->right=BST_insert(root->right,new_review);
    }
    return root;
}
```

What's happening in the function above?

It receives a pointer to the a '*BST_Node*' which is the ***root*** of a ***BST***. Keep in mind that any subtree of a ***BST*** is also a ***BST***! So, many different nodes can be thought of as being ***root***, depending on what chunk of the ***BST*** you're looking at! The figure below illustrates this.

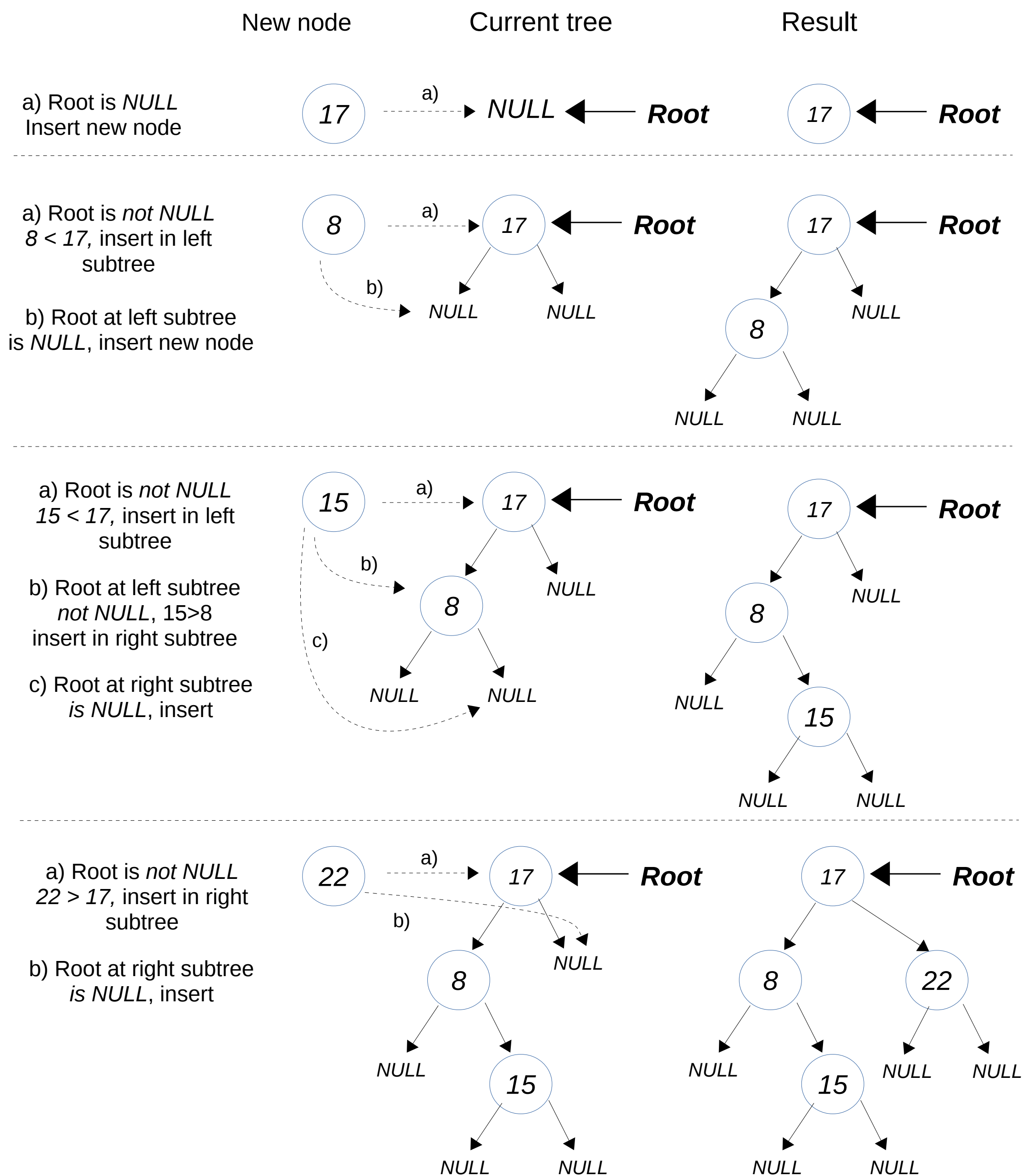


*Drawing 3: Every subtree in a **BST** is also a **BST**. The figure above shows the different subtrees (rooted at different nodes) in a sample **BST**. Note that each leaf node has **two empty subtrees** as children.*

Given the ***root*** of some BST (it can be the whole tree or a subtree thereof), the insert function checks if the ***root*** is ***NULL***. In this case the tree is empty, and the node we are inserting becomes the root.

If the ***root*** is not ***NULL***, the (sub)tree is not empty, so the insert function determines whether the new review should be in the ***left subtree*** or the ***right subtree***, and proceeds to insert the review in the corresponding subtree (using the corresponding node as ***root***). Eventually, the insert function will find an empty branch where to insert the new node, and because at each step it is testing what the correct subtree is for the new review based on how its '*restaurant_name'* compares to other reviews in the tree, the new review will be inserted at the correct location to preserve the strict ordering of keys in the ***BST***.

The figure below shows a few examples of the insertion process for a ***BST*** that contains only integer values. The process is identical for our ***BST*** with restaurant reviews except for the reviews we are comparing restaurant names rather than ints.



Exercise: Starting with the last BST in the diagram above (after 22 is inserted), list the steps carried out, and draw the resulting tree, after inserting:

19, 4, 1, 6 , and 21 (in that order)

Revisiting the complexity of search in a BST

We know that the complexity of searching for a key in a BST depends on the height of the tree, and we also know that on a ***full BST*** the height of the tree is $\lceil Log_2(N) \rceil$, however, given a randomly ordered set of keys inserted in sequence into a ***BST***, it is rarely the case that we will end up with a full ***BST***. ***The shape of the BST, and hence its height, depends on the order in which keys are inserted***.

If the order of the keys is more-or-less random, we can expect that insertions will be more-or-less equally distributed among the left and right subtrees at any level in the tree. However, if the ordering of the keys is not random, we will quickly get in trouble.

Exercise: Draw the BST that results from inserting the following keys in sequence:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10

Question: What is the height of the resulting BST? Generalizing to a set of *N initially sorted keys* inserted in sequence into the tree, what will be the height of the BST?

Question: What can you deduce from the above regarding the ***worst-case search complexity on a BST***?

Question: What does the shape of a BST resulting from inserting keys in sorted order remind you of?

The full analysis of search complexity on a ***BST*** is beyond our scope here, you'll likely have a closer look next year in B63. But for now, you should keep in mind that the ***O(Log(N))*** search complexity we discussed for ***BST***s is only ***the average case complexity***. Under the assumption that keys are randomly ordered when inserted into the tree. The ***worst case complexity*** is another matter altogether, and we should keep that in mind when deciding whether or not to use a ***BST*** to store and efficiently access a large collection of items.

Before moving on to other operations on ***BST***s, it's worth pointing out that there are several types of trees such as ***AVL-trees, B-trees, and 2-3-4 trees*** that have the property that they remain ***balanced*** regardless of the order in which keys are inserted. Such ***balanced trees guarantee*** that the complexity of ***search, insert,*** and ***delete*** operations remains ***O(Log(N))*** under all conditions. You will learn about these trees, how they work, and how they maintain a balanced shape in B63.

Search in a BST

The search operation in a BST looks for a specific ***query key***. Once the specified ***query key*** has been found, a pointer to the node containing it is returned. For BST that store compound data types, the search operation must know which field of the compound data type to use as key.

Let's see how this works:

```
BST_Node *BST_search(BST_Node *root, char name[1024])
{
      // Look up a restaurant review by restaurant name

      if (root==NULL)   return NULL;      // Tree or sub-tree is empty

      // Check if this node contains the review we want, if so, return
      // a pointer to it
      if (strcmp(root->rev.restaurant_name, name)==0)
            return root;

      // Not in this node, search the corresponding subtree
      if (strcmp(name, root->rev.restaurant_name)<=0)
      {
            return BST_search(root->left,name);
      }
      else
      {
            return BST_search(root->right,name);
      }
}
```

The implementation is nearly identical to that of the insert operation. It looks for a key matching the query in the current node, and if not found, searches the corresponding subtree depending on how the query compares to the key in the current node. The search ends when we find the key we want, or we reach an empty branch.

At this point it becomes important to talk about how to handle ***duplicate keys***. The definition of the ***BSTs*** does not forbid the existence of duplicate values in the tree. However, the behaviour of search becomes badly defined if duplicates are present – that is because there is no principled way of deciding which of the duplicate nodes the user actually wants (this was also am issue with our linked-lists in the previous Unit!).

Question: Suppose we have multiple reviews for the same restaurant. *Which* of them would be returned by the search function as defined above?

Because of the ambiguity that is introduced by the existence of duplicate keys, ***databases*** rely on the use of ***unique identifiers*** for data items. Examples you use regularly include ***student number***, ***social insurance number***, ***product bar codes***, etc. These are generated for, and associated with each item in a collection to ensure that we always have a way of identifying any particular entry with no ambiguity.

If no unique identified has been generated, it is often possible to create one from a ***combination of data fields*** from the item's information. For example, with a collection of music records, we could uniquely identify any given one by checking ***all of the following fields match a query:*** Title, Artist, Label, Year.

Database Primary Keys have the constraint of being unique, and in our ***BST***s (or linked-lists, or any other data structure intended to support search on a collection), we must enforce the use of unique keys to identify items. You'll learn all about how primary keys are designed and used in the database course, C43.

Very importantly: Once we have determined what we will use as a unique key to locate items in our collection, the insert operation must enforce that ***no duplicate records are added to the BST***. That means that if we attempt to insert an item whose unique key matches an already existing one, the insertion operation will not be carried out.

Updating an item in the collection

The update operation is simply a search followed by an update to the desired data fields in the item. ***However*** it should be noted that ***no updates are allowed on the data value(s) used as key*** for the item. Updating the key would likely mean that the BST property of having keys ordered inside the tree would no longer hold, and would therefore break search, insert, and delete operations.

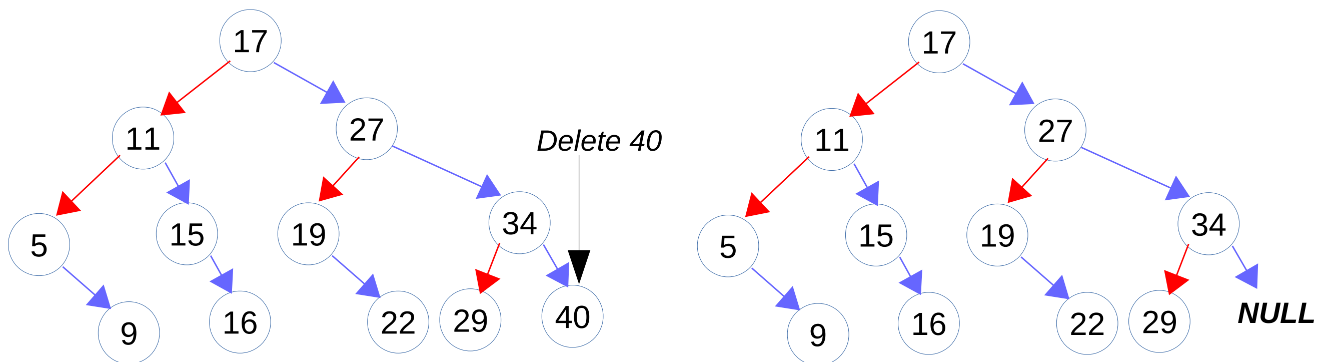
Deleting items from the BST

The *deletion* operation is a bit more interesting than both *search* and *insert*. In a linked-list, insertion involved finding the item to be deleted along with its predecessor, and then linking the predecessor and successor nodes to preserve the linked structure of the list – then deleting the desired item node.

For BSTs it's more complicated because we must ensure that after deletion the BST property still holds everywhere in the tree.

There are three cases we must consider for deletion:

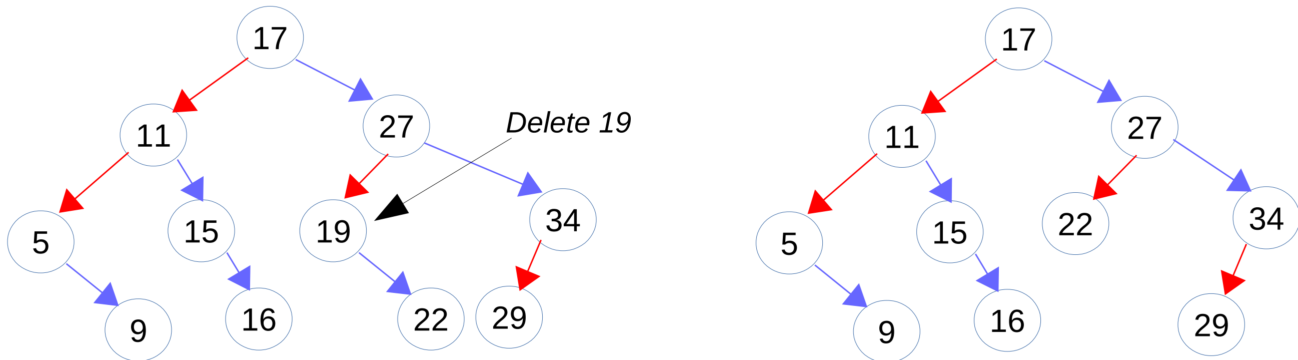
a) The item to be deleted is a leaf node (no children). In the example below, suppose we want to delete the node that contains '*40*'.



All that we need to do is set corresponding paretent node's link to ***NULL***. In the case above, *'40'* is the *right child of '34'*, so we set the corresponding link to ***NULL*** and delete (release memory) for the

node containing '*40*'.

b) The item to be deleted has ***one child only***. In which case we simply link the node's child to the node's parent and delete the node. In the example below, deleting '*19*' means that we link *'22'* to *'27'* and then release the memory used by the node with *'19'*.



The first two cases can be handled by the following code:

```
BST_Node *BST_delete(BST_Node *root, char name[1024])
{
      // Remove a review for a restaurant whose name matches the query
      // Assumes unique restaurant names!

      BST_Node *tmp;

      if (root==NULL)   return NULL;      // Tree or sub-tree is empty

      // Check if this node contains the review we want to delete
      if (strcmp(name,root->rev.restaurant_name)==0)
      {
            if (root->left==NULL && root->right==NULL)
            {
                  // Case a), no children. The parent will
                  // be updated to have NULL instead of this
                  // node's address, and we delete this node
                  free(root);
                  return NULL;
            }
            else if (root->right==NULL)
            {
                  // Case b), only one child, left subtree
                  // The parent has to be linked to the left
                  // child of this node, and we free this node
                  tmp=root->left;
                  free(root);
                  return tmp;
            }
            else if (root->left==NULL)
            {
```

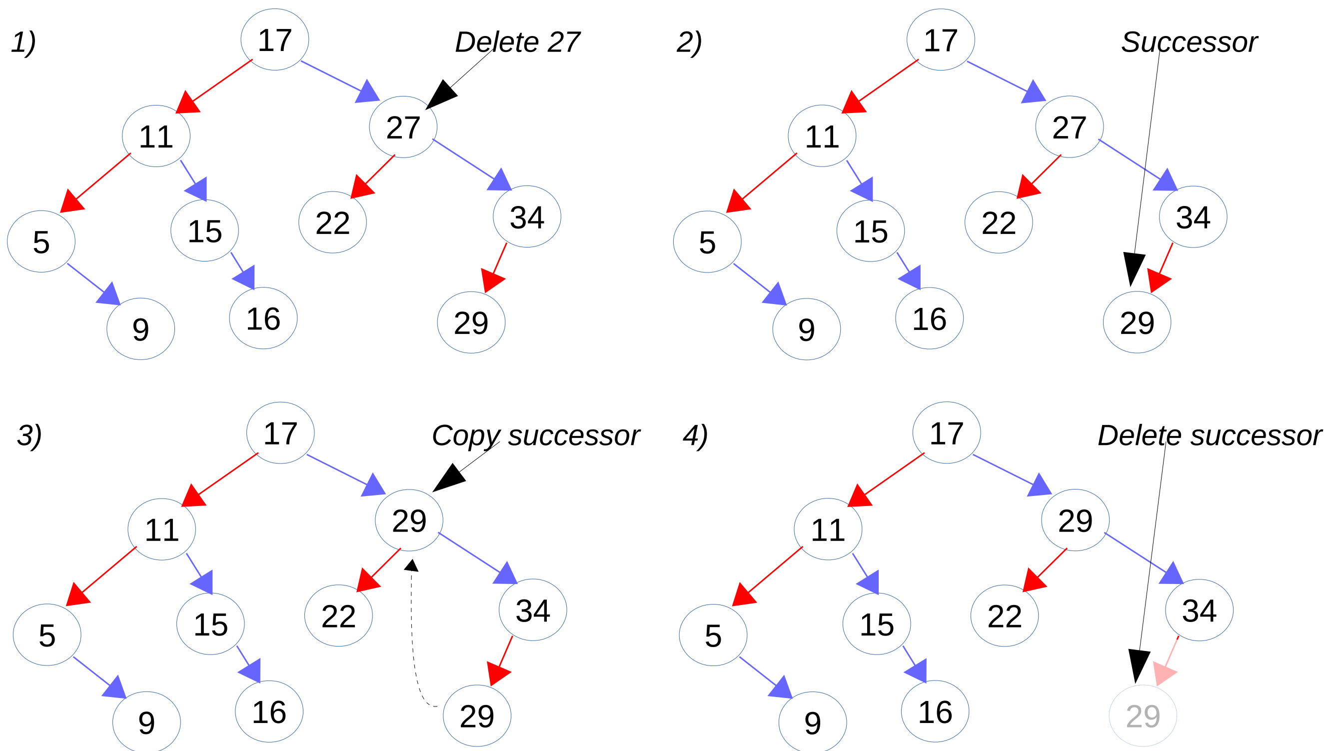
```
                // Case b), only one child, right subtree
                // The parent has to be linked to the right
                // child of this node, and we free this node
                tmp=root->right;
                free(root);
                return tmp;
            }
            else
            {
                // Case c), two children.
                // You will implement this for your exercise!
              return root;
            }
        }

        // Not in this node, delete on the corresponding subtree and
        // update the the corresponding link
        if (strcmp(name, root->rev.restaurant_name)<=0)
        {
            root->left=BST_delete(root->left,name);
        }
        else
        {
            root->right=BST_delete(root->right,name);
        }
        return root;
}
```

Now, for the last case:

c) The node to be deleted has ***two children***. This is a bit more challenging. The way deletion works when the node we want to remove has two children is as follows: We find the ***successor*** to the node we want to delete – this is the node whose ***key follows in order the one in the node we are deleting***. Another way to think about it is: ***the successor is the smallest key in the right subtree***.

Once the successor has been identified:
- Copy the ***data*** from the ***successor node to the node we are deleting***.
  - If the node contains a compound data type, copy the whole compound data type
- Then ***delete the successor node from the right subtree.***
  - This can again be either of the a), b), or c) cases above.



As you can see, the actual ***node*** where '*27*' was stored didn't go anywhere, and we did not need to re-link anything! What we did is replace the data in the node with the *successor's data* and then delete the successor node.

Exercise: Figure out what the tree would look like after deleting the following keys in sequence (draw the tree after each deletion):

9, 22, 29, 11

Exercise: Complete the implementation of the *BST_delete()* function above to implement case c). You will need to write code to find the successor so you can copy its data over and so you know what do delete from the subtree.

Tree Traversals

There is one more set of operations on ***BST***s that we need to discuss – these are called *tree traversals*. A tree traversal constists of *visiting each node in the **BST** in a pre-specified order.* We use tree traversals for different purposes, including:

- Printing data for nodes in the ***BST***
- Computing aggregate statistics for data in our collection
- Updating information when the update applies to all nodes (e.g. suppose our ***BST*** contains item prices for a store, and there is a store-wide sale with discounts applied to everything).

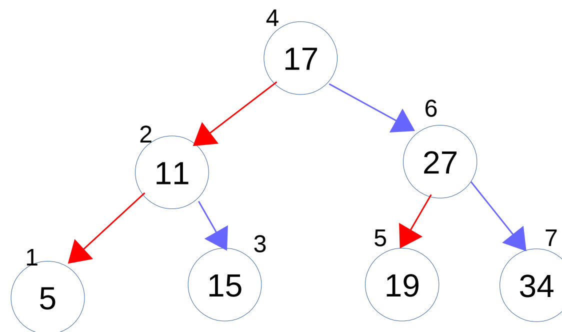
- Deleting the ***BST*** once our program is done

There are three main types of traversals, distinguished by the order in which nodes are visited:

- ***In-order traversal***, which is possibly the most common and specifies that tree nodes are visited in this order for any given subtree:

1) Traverse left sub-tree ***in-order***
2) Root – apply desired operation on the node
3) Traverse right sub-tree ***in-order***

The example below shows the order in which nodes in a tree would be visited during an in-order traversal.



If at each node, the operation we carry out is simply to print the key at the node, the in-order traversal would produce:

5, 11, 15, 17, 19, 27, 34

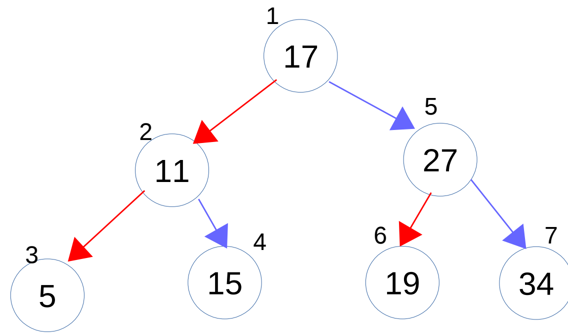
Which gives you an *ordered list* of the *keys in the BST.* Hence the name of the traversal.

Exercise: Write a function to print the restaurant reviews in alphabetical order of restaurant name.

- ***Pre-order traversal,*** for this type of traversal the order in which nodes are visited is given by:

1) Root – apply the desired operation on the node
2) Traverse in ***pre-order*** the left subtree
3) Traverse in ***pre-order*** the right subtree

The tree below shows the order in which nodes would be visited in ***pre-order***.



This time, if the operation at each node was to print the key, we would obtain the following list:

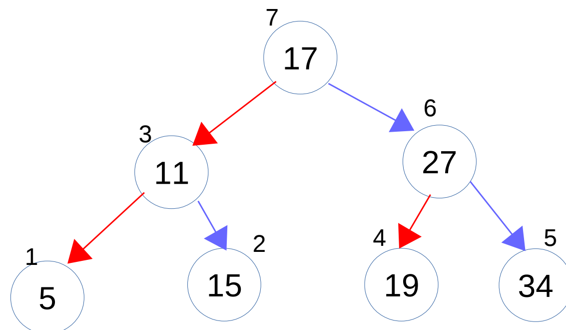
17, 11, 5, 15, 27, 19, 34

This may not seem like a very interesting list, however, ***pre-order traversal*** is very useful in applications such as compilers and natural language understanding, where the structure of a chunk of code, a sequence or mathematical operations, or parts of speech in a sentence is represented by a tree structure. It is also the traversal order we need to use if we want to create a copy of a ***BST***.

- ***Post-order traversal,*** specifies that the nodes in the tree must be visited in the following order:

1) Traverse in ***post-order*** the left subtree
2) Traverse in ***post-order*** the right subtree
3) Root – apply the desired operation on the node

The tree below shows the order in which nodes would be visited by a ***post-order traversal.***



With ***post-order*** traversal, we would obtain the following list of keys:
5, 15, 11, 19, 34, 27, 17

Once more, the list doesn't look particularly interesting to the eye, but ***post-order traversal*** has applications in parsing, and is also the order in which we need to visit nodes when *deleting a BST*.

Exercise: Draw the tree that results from inserting the following keys in sequence:

57, 25, 69, 16, 35, 59, 71, 72, 73, 9, 3, 14, 99

Then list the entries in the order that would be produced by *in-order, pre-order, and post-order traversals.*

Exercise: Write a function to delete the ***BST*** for restaurant reviews, ensuring all memory allocated to nodes of the ***BST*** is released.

Exercise: Complete your implementation of a little restaurant review database by adding an interactive loop in main() that allows you to input new reviews, search for specific reviews, or delete existing ones.

7.- Have we solved our problem?

At this point we have a good understanding of what ***BST***s can do, and how to implement them, but we have yet to determine whether all this work has helped us solve the problem of finding a more efficient way to store, organize, and access a large collection of information. We may suspect that the ***BST*** is likely to give us faster search because of its average search complexity being better than the linked-list's average search complexity, but we still have to tie a few loose ends:

a) Time it takes to build the ***BST*** vs. time needed to build a linked list

- For ***N*** items, the linked list can be built in ***O(N)*** time – each item is added at the head so no traversal is needed. That's pretty good!
- For the same ***N*** items, the BST can be built in ***O(N Log(N))*** time ***on average*** since insertions always happen at the ***bottom of the tree***, and the tree has height ***O(Log(N))***.

Advantage: Linked list

b) Time it takes to build a ***BST*** vs. time needed to sort an array (so we can use binary search)

- Using *quicksort()* an array can be sorted in ***O(N Log(N))*** time ***on average***.
- As we saw above, the ***BST*** can be built in ***O(N Log(N))*** time ***on average***.

Advantage: None – it's a tie in terms of complexity

c) Search complexity

- Sorted array: ***O(Log(N)) worst case*** using binary search.
- Linked list: ***O(N) both average and worst case***.
- ***BST***: ***O(Log(N)) average***, ***O(N) worst case***.

Advantage: BST and sorted array on average, sorted array in ***worst case***.

d) Storage use

- Sorted array: Fixed size, not suitable for growing/shrinking collections
- Linked list: Space usage is ***O(N)*** – one node per item
- ***BST***: Space usage is ***O(N)*** – one node per item

Advantage: BST and linked-list, they only use space for items actually added to the collection.

So what do we choose?

The point of going through all of this work is to help you see that there is a fair number of factors you have to consider when choosing how to store and organize a collection of data. Up to this point we have 3 major ways of storing items: arrays, linked-lists, and ***BST***s, and as shown above, each one of them has advantages and disadvantages. So how are we to choose?

a) Consider how large your collection is going to be:

- ***For very large collections***, you want to choose the data structure that gives you the best ***Big O*** complexity for common operations like *insert*, *delete*, and *search*.

    - Consider the space requirements: If items have small memory requirements (e.g. just numeric data, like ints or floats, or small compound data types) it's not unreasonable to pre-allocate a large array even though entries in it may go un-used. If, on the other hand your items have a large memory footprint (e.g. compound data types with lots of data, or multi-media content like images, sound clips, etc.) you definitely want a data structure that allocates memory ***only for as many items as you have in the collection at a given time.***

- ***For smaller collections***, it will depend on whether you favour ease of implementation vs. the performance of your data structure for common operations.

b) Consider what kind of operations will be performed on the collection's items:

- If you will mostly perform operations over the entire set, choose a linked list or an array (it doesn't have to be sorted). An example of this is the 3D point meshes used in computer graphics, which may contain millions of points, but we don't perform individual point look-ups, and instead render the whole mesh. These are often kept in arrays.

- If *search, insert,* or *delete* will be frequent, then you should favour a data structure that gives you the best ***Big O*** complexity for these operations.

You do not have to settle for ***one data structure***. Databases organize large amounts of information in the form of ***tables*** – these are basically huge arrays, where each row contains all the data fields for one item in the collection. They are stored ***on disk*** and are ***unsorted***. Separately from the table that contains the actual data, the database keeps ***an index*** in the form of a tree (typically a ***B-tree*** or a variation thereof) which contains ***search keys*** the user will be running queries on. Each node in the ***index*** contains the location in the ***original table*** where the information for the item matching the query key can be found.

An additional advantage of this scheme is that the database can have multiple indexes for the same table: For example, for student records in ACORN, there could be an index organized by student number, and one organized by last-name/first-name. Each index allows the database to efficiently search for records based on different common query terms.

Always remember: Whenever you are designing a solution for storing, organizing, and accessing information, you must consider the size of the collection, how it will be used, what operations will be run on it, and then think about all the ADTs and data structures you know, their complexity class for common operations, and how you could use them to build the most efficient data storage and manipulation solution. ***Look forward to B63 for learning a whole lot more about ways to store, organize, and search for information***, including balanced trees, hash tables, heaps, graphs, and combinations of these.

Does it all work in practice?

It's worth taking a look at whether all we have learned here actually works out in practice. The graphs you will find below compare the performance of our three storage solutions for the task of organizing and searching through a large collection of integers. Our three methods are:

- A sorted array + binary search
- A linked list
- A ***BST***

We will test them on collections of increasing size, and on each collection, we will run 10,000 queries for randomly chosen keys (to check how the search performance compares across these data structures).

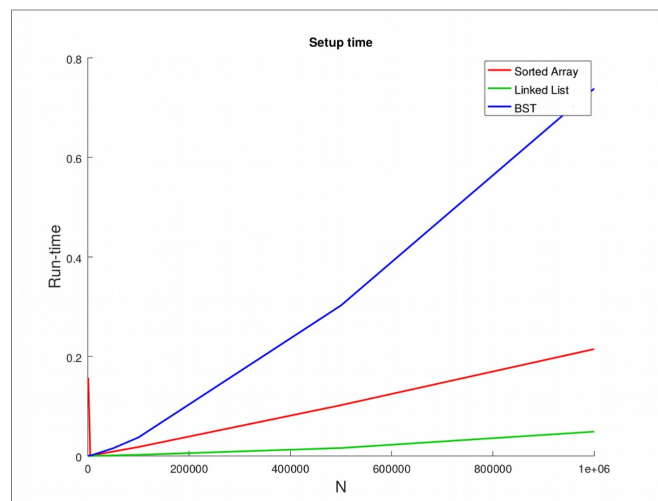
You will recall that one worry with ***BST***s is that the worst case complexity is ***O(N)*** for *search*, *insert*, and *delete*. It's important to figure out whether the performance of the data structures over multiple tests, on different sets of items, may indeed change so much that we ***should not rely on average case complexity*** when deciding which of them to use. In order to study this, we will run 50 tests for each collection size, each with different randomly-generated keys. It's a relatively small sample but it should at least give us an indication of whether our data structures perform as we would expect on different input data sets.

Finally, we show the run-time in three separate ways:

- Set up time: Time it takes to insert the data into the collection (for the array this includes the time it takes to run quicksort()).
- Search time: The time it takes to run 10,000 queries once the collection is stored
- Total time: The sum of both, giving a complete picture of how each data structure performs.

Important note on the graphs below: Please be very attentive – the ***x-axis is not to scale!*** each tick in the *x* direction corresponds to ***10 times more items than the previous one***. If we showed the graph to scale, it would spill off the page and run into the next room! Keep this in mind as you think about what the graphs are showing.

Set-up time: Time taken to insert ***N*** items into the data structure – for the array this includes time taken by quicksort().



The graph shows the set-up time for all three data structures. As we expected from our analysis above, the linked-list has the advantage, with the set-up time being only ***O(N)***. Both the sorted array and the ***BST*** take significantly longer.

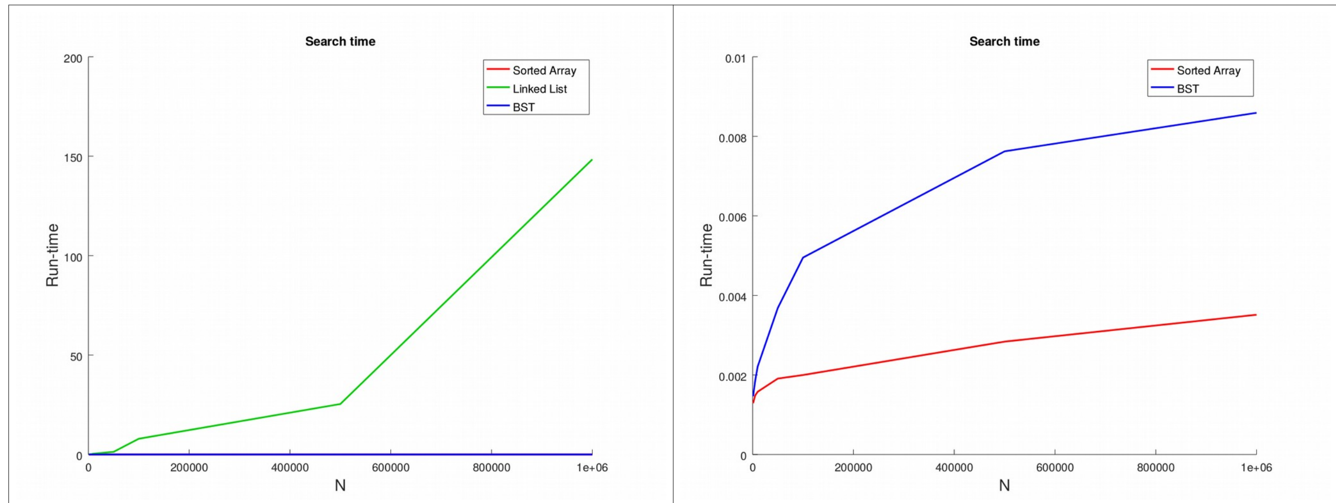
Conclusion: Set-up time in practice agrees with our analysis of the behaviour of each data structure in the sections above.

Question: There is a weird spike on the set-up time for the sorted array at the smallest number of items (it's not at zero, that is just an effect of the scale of the x-axis). What could possibly explain this strange behaviour?

Search Time

Below, on the left, you can see the time it took to run 10,000 searches on each of the data structures. Averaged over the 50 different runs with different ordering of random keys. As you can

easily see, the performance of the linked-list is much worse than the array and ***BST***. This is also as expected, the average ***O(N)*** search complexity of the linked-list can not compete with the average ***O(Log(N))*** search complexity of the ***BST***, and the guaranteed ***O(Log(N))*** search complexity of binary search. The plot on the right shows only the ***BST*** and sorted array search times, for comparison.



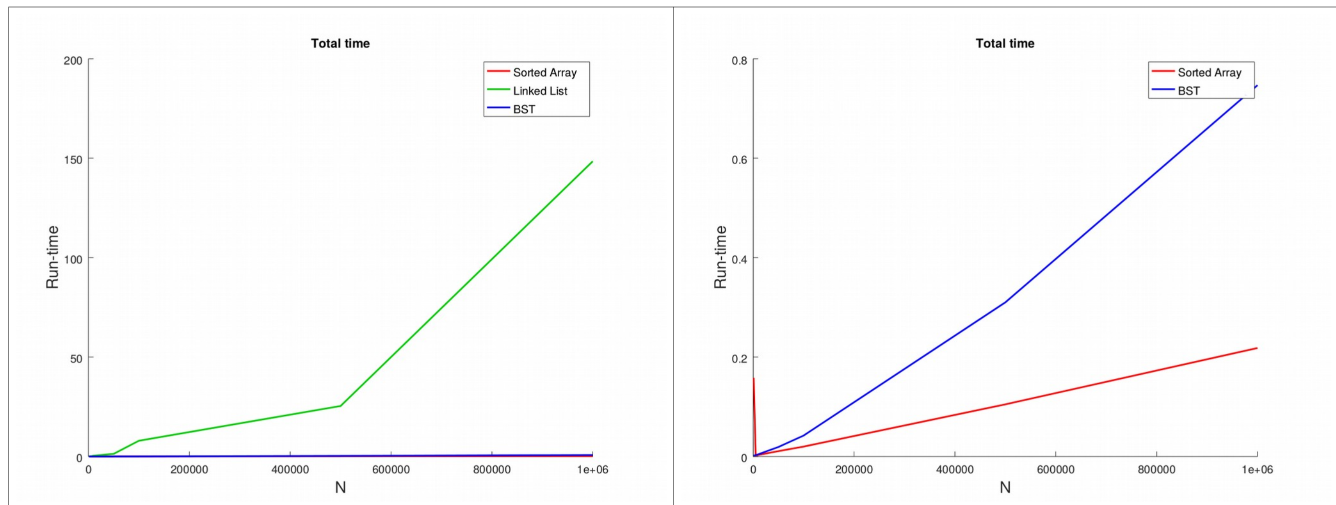
The graphs above show behaviour that is consistent with what we expected. The ***BST*** and sorted array perform similarly well (the shape of the curves for run-time looks fairly close, except for some constant factor related to the overhead in managing the ***BST*** vs. the array). We can also see that the ***BST***'s performance does not appear to show any slow-down indicative of hitting the worst-case ***O(N)*** for search complexity (in which case we should expect it to move toward the green curve of the linked-list in the plots above).

Conclusion: Search time in practice agrees with our analysis of complexity in the sections above.

Total time

The graphs below show the total time taken by each of the data structures to carry out the 10,000 queries. It's the sum of the set-up time and the search time.

Once again, the graph on the left shows the total time for all three data structures, and clearly indicates that if we expect to be doing a lot of searching for items in our collection, the linked-lists are ***not*** the data structure of choice. Both the ***BST*** and sorted array perform similarly well – similar shaped curves with the difference likely very close to a constant factor explained by the overhead of maintaining and traversing the ***BST***, compared to almost immediate access to any entry in the array.



One more observation: The shape of the total-time curves indicates that the ***total time*** for the ***BST*** and sorted array is ***dominated by the set-up time*** at least for the number of queries that we performed on the database. This should not be surprising. The set-up time for both the sorted array and the BST is ***O(N Log(N))*** (for the ***BST***, remember, this is only ***on average***). The search time, on the other hand, is only ***O(m Log(N))*** where ***m*** is the number of queries we run on the data structure. Since ***m=10,000***, the search time is expected to be smaller than the set-up time for the largest collections.

Conclusion: Total search time agrees in practice with our analysis of the behaviour of the data structures from the previous sections.

Final Thoughts

We have spent some time carefully studying the problem of how to make search more efficient for large collections. Along the way we have studied the problem of measuring, characterizing, and reasoning about the ***complexity*** of algorithms, problems, and data structures. We have learned that the complexity results we can derive from studying a given solution are clearly and directly translated to run-time when we implement these solutions, and we have thought about how and when we should use the different ***ADT***s we have studied up to this point.

Keep in mind that the theoretical analysis of complexity is only one layer of the complex problem of figuring out the most efficient way to carry out some task. You need to learn about computer architecture, how modern CPUs work, and how to optimize programs for maximum efficiency if you really want to write the best (most efficient, fastest) software to solve any given problem. Don't forget to keep what you learned here in mind when you take Computer Organization (B58), Algorithms and Data Structures (B63), and Embedded Systems (C85), as these three courses will provide you with deeper understanding of this important problem.

As for the ***worst-case complexity***. The analysis above may lead you to believe you don't have to worry about it if your data structure has a good ***average-case*** complexity. Indeed, for most applications

you will find you can get excellent results from using data structures such as ***BST***s, and sorting algorithms such as *quicksort()*. ***However:*** For safety-critical applications, or for real-time applications, examples of which include control software for medical equipment, electrical power generation, transportation (aircraft flight control), robotics, manufacturing, and so on; ***you can not afford to use a data structure whose worst-case performance is bad***. The point being that even if you need to be very unlucky to hit the input that triggers worst-case or close to worst-case performance, you can not afford to take that risk. Applications of this kind require ***guaranteed performance bounds*** from all the data structures and algorithms involved in their software. So, keep this in mind as you move to B63 and C85 where these issues will be discussed at length.

Additional Exercises

Ex0 – Draw the BST that results from inserting the following keys in sequence into an initially empty BST. Keys are sorted alphabetically.

“Iron Man 3”, “Black Panther”, “The Avengers”, “Captain America”, “Iron Man 2”, “Aquaman”, “Batman Returns”, “Thor”, “Spider Man: Into the Spiderverse”

Ex1 – Draw the BST after we delete “Aquaman”

Ex2 – Draw the BST after we delete “Iron Man 3”

Ex3 – What is the list of movies generated by a ***post-order traversal*** of the BST from Ex2?

Ex4 – It is highly likely that a BST whose keys are movie titles will eventually run into the problem that there are multiple entries with the same key. As we discussed above this is really not a great situation. ***Give at least 3 different suggestions regarding how we can build a BST where entries are organized by movie title, yet, there are no duplicate keys.***

Ex5 – Which of the following has the ***lowest*** complexity for ***large values of N***?

a) 250000*Log(N)
b) .001*N
c) .000001*N^2
d) 5000*Log(N^2)

Ex6 – Which of the following has the ***lowest*** complexity for ***small values of N***?

a) 5.25*N
b) 1.11*N
c) 5*Log(N)
d) 2.1*Log(N^2)

Ex7 – Recall that in the discussion above we specified that updates to the ***key*** values in nodes for a

BST is not allowed because it can break the ***BST*** property. However, in practice we may come upon a situation in which we *have to update a data field used as key*. List in the space below the steps we could take to accomplish this, without breaking the ***BST*** property in the tree (pseudocode is fine, no need to write C code for this). *Hint:* Consider all available ***BST*** operations.

Ex8 – Implement *quicksort()* from the description in P.16. You can consult on-line resources as well.