Notes for ECE 36800 - Data Structures and Algorithms

Shubham Saluja Kumar Agarwal

April 11, 2024

These are lecture notes for spring 2024 ECE 36800 at Purdue. Modify, use, and distribute as you please.

Contents

Course Introduction

Provides insight into the use of data structures. Topics include stacks, queues and lists, trees, graphs, sorting, searching, and hashing. The learning outcomes are:

- Advanced programming ideas, in practice and in theory
- Data structures and their abstractions: Stacks, lists, trees, and graphs
- Fundamentals of algorithms and their complexities: Sorting, searching, hashing, and graph algorithms
- Problem-Solving

Introduction to Data Structures & Algorithms

Data Structures are methods of organizing information for ease of manipulation. Examples:

1. Dictionary
2. Check-out line or queues
3. Spring-loaded plate dispenser or stacked
4. Organizational Chart or tree

These are associated with methods known as algorithms to be manipulated
Algorithms are methods of doing something. Examples:

1. Multiplying two numbers
2. Making a sandwich
3. Getting dressed

The topics of interest within them are:

- Correctness
- Efficiency in time and space

Time Complexity Analysis

The questions to be asked about an algorithm are the following:

- Is it correct?
- Is it as fast as possible?
- How many machine instructions (in terms of n) does it take?

Let us take the following algorithm to add the numbers form 1 to n:

```
total = 0;
for (i=1:n)
    total = total + i;
return total
```

The cost will be:

Cost	Frequency	Function
C_1	1	Assign initial value
C_2	n+1	For loop iterations and exit
C_3	n	Number additions
C_4	1	Return value

The total is then:

$$C_1 * 1 + C_2(n+1) + C_3(n) + C_4(1) = (C_2 + C_3)n + (C_1 + C_2) + C_4$$

However the $O(n)$ will only be n, as the constants and coefficients of these will be deprecated, as we will come to understand in more detail as this topic continues.

Let us take another example of some code that has a

$$\begin{aligned} T(n) &= n^2 + 10^7 n + 10^{10} \\ T(10^{11}) &= 10^{22} + 10^{18} + 10^{10} \\ T(2*10^{11}) &= 4*10^{22} + 2*10^{18} + 10^{10} \\ \implies \frac{T(2*10^{11})}{T(10^{11})} &\approx 4 = \left(\frac{2*10^{11}}{10^{11}}\right)^2 \end{aligned}$$

This goes to show that this algorithm has an $O(n) = n^2$, and all coefficients and lower order terms that are a part of the complexity are largely irrelevant for large n values. This is why this is called **asymptotic notation**.

Another example of a simple algorithm is

```
total = 0;
for (i=1:n):
    if (((i*i%3)==0)||((i*i%7)==0)):
        total = total+i*i;
return total;
```

Which has a cost table that looks like the following:

Cost	Frequency	Function
C_1	1	Assign initial value
C_2	n+1	For loop iterations and exit
C_3	n	Number of $i\%3$ comparisons
C_4	$n - \lfloor \frac{n}{3} \rfloor$	Number of $i\%7$ comparisons
C_5	$\lfloor \frac{n}{3} \rfloor + \lfloor \frac{n}{7} \rfloor - \lfloor \frac{n}{21} \rfloor$	Number of additions
C_6	1	Returning value

It can be noted that $O(n) = n$ for this function, despite all the other complexities in the algorithm. However, it is important to know how to calculate $T(n)$ as well.

Now, let us look at something more complicated, matrix multiplication of two lower triangular matrices.

```
for (i=1:n):
    for (j=1:i):
        C_ij = 0;
        for (k=j:i):
            C_ij = C_ij+A_ik*B_kj
return C
```

This has a cost table that looks like the following: Finally, we can

Cost	Frequency	Function
C_1	n+1	First loop
C_2	$\sum_{i=1}^{n}(i+1)$	Second loop
C_3	$\sum_{i=1}^{n}\sum_{j=1}^{i}1$	Number of assigns
C_4	$\sum_{i=1}^{n}\sum_{j=1}^{i}(i-j+2)$	Third loop
C_5	$\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=j}^{i}1$	Number of assigns to matrix
C_6	1	Returning value

analyze an example that has logarithmic complexities.

```
i=2;
k=0;
while (i<n){
    i=i*i;
    k=k+1;
}
return i;
```

Which has a cost table that looks like the following:

Cost	Frequency	Function
C_1	1	Assign i
C_2	1	Assign k
C_3	$\lceil\log_2(\log_2(n))\rceil+1$	Number of while loop iterations
C_4	$\lceil\log_2(\log_2(n))\rceil$	number of assigns of i
C_5	$\lceil\log_2(\log_2(n))\rceil$	Number of k assigns
C_6	1	Returning value

It can be noted that if line three was instead changed to

```
while (i<=n){
```

The table will instead be:

Cost	Frequency	Function
C_1	1	Assign i
C_2	1	Assign k
C_3	$\lceil \log_2(\log_2(n+1)) \rceil + 1$	Number of while loop iterations
C_4	$\lceil \log_2(\log_2(n+1)) \rceil$	number of assigns of i
C_5	$\lceil \log_2(\log_2(n+1)) \rceil$	Number of k assigns
C_6	1	Returning value

As the loop break condition changed from $i \geq n$ to $i \geq n+1$ by simply changing.

Insertion and Shell Sort

Sorting is necessary to process items in sorted order. It speeds up the location of items, finding identical items, etc.
It is good to know that in real life, what is sorted is in fact the pointers of these structs, as the movement of structs have higher memory requirements.

Insertion Sort

Inserts an item into a sorted array. Compares the item with items in the sorted array, and if they are in the incorrect order, they are swapped. This is continued until everything has been successfully sorted.
The code to sort n integers in an array r looks like this:

```
for (j=1:n-1){
    for (i=j:1){
        if (r[i-1]>r[i]){
            swap(r[i-1], r[i]);
        }
        else{
            break;
        }
    }
}
```

This is suboptimally inefficient due to the restriction of only swapping with neighbors, directly. However, it can be made even more efficient using the following algorithm:

```
for (j=1:n-1){
    temp = r[j];
    for (i=j:1){
        if (r[i-1]>temp_r){
```

```
                r[i] = r[i-1];
            }
            else{
                break;
            }
        }
        r[i] = temp_r;
    }
```

This allows us to "move" items down without constant comparisons, saving us some assignments.
This can also be implemented using while loops, and thus avoiding break:

```
    for (j=1:n-1){
        temp=r[j];
        i=j;
        while (i>0 and r[i-1]>temp){
            r[i] = r[i-1];
            i-=1;
        }
        r[i]=temp_r;
    }
```

This has the following cost table in the best case: Which has a com-

Cost	Frequency	Function
C_1	n	For loop iterations
C_2	n-1	Assign temp
C_3	n-1	Assign i
C_4	n-1	It is checked once per iteration
C_5	0	Number of r[i] exchanges
C_6	0	Number of decreases of i
C_7	n-1	Assign r[i]

plexity $O(n) = n$ And the following in the worst case: Now, we will

Cost	Frequency	Function
C_1	n	For loop iterations
C_2	n-1	Assign temp
C_3	n-1	Assign i
C_4	$\frac{(n+2)(n-1)}{2}$	Number of time the while loop is checked
C_5	$\frac{(n)(n-1)}{2}$	Number of r[i] exchanges
C_6	$\frac{(n)(n-1)}{2}$	Number of decreases of i
C_7	n-1	Assign r[i]

learn how to calculate the average performance of an algorithm like insertion sort.

Let us take a random j^{th} item. The probability of it not needing to be moved is $\frac{1}{j+1}$. And it will need a certain some number between 0 and j exchanges to get to its rightful position if not. This leads the expected total number of exchanges to be $\sum_{i=0}^{j} \frac{i}{j+1} = \frac{j}{2}$. Once we reach the $(n-1)^{th}$ element, this is $\frac{1}{2}\frac{n(n-1)}{2} \approx \frac{n^2}{4}$.

Average performance is seldom calculated for the intents and purposes of this course.
There are still some inefficiencies in insertion sort that can be improved by using sentinels.

```
for (j=n-1:1){
    if (r[j]<r[j-1]){
        swap(r[j], r[j-1]);
    }
}
for (j=2:n-1){
    temp=r[j];
    i=j;
    while (r[i-1]>temp){
        r[i] = r[i-1];
        i-=1;
    }
    r[i]=temp_r;
}
```

By moving the smallest item to the beginning, we can avoid the (i>0) condition, slightly increasing efficiency.

Shell Sort

This improves insertion sort by allowing for swaps along larger distances between elements.
If we did 7-sorting and 3-sorting:
We would start with 7 subarrays with at most $\lceil \frac{n}{7} \rceil$ elements. These subarrays would need to be sorted within themselves.
We would then go over to having 3 subarrays with at most $\lceil \frac{n}{3} \rceil$ elements. These subarrays would once again need to be sorted within themselves.
Finally, we would conduct regular insertion sort. The complexity of shell sort changes based on the selected sequence.

- $1, 3, 7, 15, \ldots, 2^k - 1, \ldots$ has a complexity of $O(n^{1.5})$
- $1, 4, 13, \ldots, 3h(k-1), \ldots$ also has a complexity of $O(n^{1.5})$
- $2^p 3^q$ has a complexity $O(n(\log(n))^2)$

The algorithm of shell sort is the following:

```
for (j=k:n-1){
    temp=r[j];
    i=j;
    while (i>=k and r[i-k]>temp){
        r[i]=r[i-k];
        i=i-k;
    }
    r[i]=temp;
}
```

To prove the complexity of the 2^p3^q complexity, we can visualize the following triangle:

$$
\begin{array}{llll}
1 \\
2 & 3 \\
4 & 6 & 9 \\
8 & 12 & 18 & 27
\end{array}
$$

which holds the values of k from the above algorithm the height and base of the triangle both have complexities of $\log(n)$, while the sorting of the subset make by each k has a complexity of n, which results in a total complexity of $n(\log(n))^2$.
Let us consider a triangle of the following form:

$$
\begin{array}{ll}
a \\
2a & 3a
\end{array}
$$

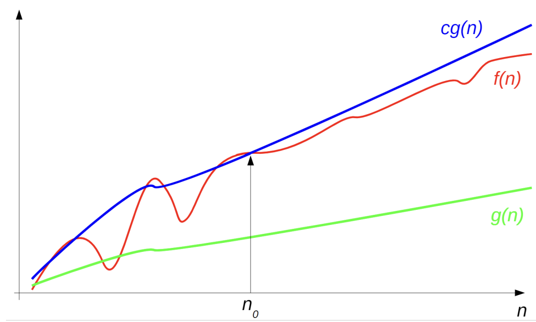
If one were to start with the largest and go to the smallest, that is, if one were to $3a$-sort, and then $2a$-sort, all numbers in the array would be at most a positions away from their correct positions.
This can be semi-trivially proven under the assumption that if an array is $3a$-sorted and then $2a$-sorted, it will still be $3a$-sorted. This will not be solved in this document as it is a homework assignment.

Asymptotic Notation

The number of instructions executed is dependent on the number of inputs. As it gets larger, the number of instructions increases too. This has been generalized and classified using asymptotic notation.

$$f(n) = O(g(n)) \iff \exists(c, n_0) \text{ such that } f(n) \leq cg(n) \quad \forall(n \geq n_0)$$

Note :If the space complexity is $O(g)$, the time complexity will be at least $O(g)$.
The graphical representation of the above definition $O(n)$ is :



The values of c and n_0 can be variable, and have any value, as long as the condition is fulfilled.
However, we are searching for $g(n)$ such that it refers to the smallest and simplest possible function of n to allow for the existence of the values of c and n_0 that will make the condition true.
A good strategy to select c is the sum of the absolute values of all the coefficients in $f(n)$.
Note: $f(n)$ is equivalent to the $T(n)$ in the Time Complexity Analysis section.
Another proof of $f(n) = O(g(n))$ is $\lim_{n\to\infty} \frac{f(n)}{g(n)} \neq \infty$.
However, it is not really necessary to go to this extent to find c as there are several easier ways to find valid values. This method is an unnecessary complication for most cases.
For $g(n)$ to be useful, it should be simpler than $f(n)$ such as $1, n, n^2, n\log(n), 2^n, n!$.
However, once we go to exponential functions or above, the algorithms cease to be useful.
Some properties of asymptotic notation are:

$$f(n) = O(f(n))$$
$$f(n) = O(g(n)) \quad \& \quad g(n) = O(h(n)) \implies f(n) = O(h(n))$$
$$f(n) = O(g(n)) \quad \& \quad g(n) = O(h(n)) \implies f(n) + g(n) = O(h(n))$$
$$f_1(n) = O(g_1(n)) \quad \& \quad f_2(n) = O(g_2(n)) \implies f_1(n) * f_2(n) = O(g_1(n) * g_2(n))$$

Hierarchy of $O(n)$ (in this section, we will show what possible asymptotic notations functions can have.):

1. $k = O(1)$
2. $k = O(\log_m(n))$

3. $k \log_m(n) = O(\log(n))$

4. $k(\log_m(n))^i = O(n)$

5. $kn = O(n)$

6. $kn \log_m(n) = O(n \log(n))$

7. $kn \log(n) = O(n^2)$

8. $kn^i = O(n^l) \quad \forall (j \geq i)$

9. $kn^j = O(d^n \quad \forall (d > 1))$

10. $kd^n = O(d^n) \quad \forall (d > 1)$

11. $kd_1^n = O(d_2^n) \quad \forall (d_1 > 1 \quad \& \quad d_1 < d_2)$

12. $kd^n = O(n!)$

13. $kn! = O(n!)$

14. $kn! = O(n^n)$

15. $kn^n = O(n^n)$

There is a notation that goes hang in hand with $O(n)$:

$$\begin{aligned} f(n) = O(g(n)) &\implies g(n) = \Omega(f(n)) \\ \exists c, n_0 \mid f(n) \leq g(n) &\implies \exists c', n_0' \mid g(n) \geq f(n) \\ &\implies \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \end{aligned}$$

And another, the Theta Notation:

$$\begin{aligned} f(n) = \Theta(g(n)) &\iff \exists (c_1, c_2, n_0) \quad c_1 g(n) \leq f(n) \leq c_2 g(n) \\ &\iff \lim_{n \to \infty} \frac{f(n)}{g(n)} \neq 0, \infty \\ &\iff [f(n) = O(g(n))] \wedge [g(n) = O(f(n))] \end{aligned}$$

And finally the little-o notation:

$$\begin{aligned} f(n) = o(g(n)) &\iff \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \\ &\iff \forall c, \exists n_0 \mid f(n) \leq cg(n) \quad \forall n \geq n_0 \end{aligned}$$

Which leads to the little-omega notation:

$$\begin{aligned} g(n) = \omega(f(n)) &\iff f(n) = o(g(n)) \\ &\iff \lim_{n \to \infty} \frac{g(n)}{f(n)} = \infty \end{aligned}$$

For most cases, if f is $O(g)$, but not $\Theta(g)$, it is $o(g)$. Exceptional cases would be like $f(n) = n^{|\sin(n)|}$, which seldom exist.
There are some implications that come from these relationships:

$$\begin{aligned}
f(n) = \Theta(g(n)) & \\
& \implies f(n) = \Omega(g(n)), f(n) = O(g(n)), \\
& \implies g(n) = \Omega(f(n)), g(n) = O(f(n)) \\
f(n) = o(g(n)) & \\
& \implies f(n) = O(g(n)) \\
f(n) = \omega(g(n)) & \\
& \implies f(n) = \Omega(g(n))
\end{aligned}$$

Linked Lists

NOTE: my notes on this section are not great, as much of the explanation was done through images and drawings upon them. If you have better notes for this section, please feel free to improve them.
A list is a linear collection of items that can be inserted and removed at any position.
Arrays have O(1) access and overwrite, although insertion or removal may be more than O(1). The size of the array is necessary to program correctly.
A linked list has the followings structure defining it:

```
typedef int Info_t;
typedef struct _Node
{
    Info_t data;
    struct _Node *next;
} Node;

typedef struct _Header
{
    Node *head;
    Node *tail;
} Header;
```

So, a linked list can be represented as the following:

$$\boxed{6 \mid \rightarrow}\ \boxed{3 \mid \rightarrow}\ \boxed{4 \mid \rightarrow}\ \boxed{\emptyset}$$

Need to have Head, without it, we cannot do anything.
One can also have the tail, if we so wanted.
The header may contain the addresses of the head, tail, and other useful information for easy access.

They have primitive operations:

- Empty: return true/false
- First: return address to first node
- Last: return address to lat node
- Insert at head: insert as the first node of the list
- Insert at tail: insert as last node
- Remove at head
- etc.

C language works by making copies. Whenever one sends information into a function, one creates a copy of that information to be operated on.
For linked list, or structs in general, this is relevant because their manipulation can be made easier by passing terms to a function in the form:

```
func(**head)
```

This allows us to point and modify the list through the address of the struct instead of a copy of it.
For example:

```
void List_insert_in_order(Node **head_addr, Node *node) {
    Node dummy;
    dummy.next = *head_addr;
    Node *curr = dummy.next;
    Node *prev = &dummy;
    while (curr && curr->data < node->data) {
        prev = curr;
        curr = curr->next;
    }
    prev->next = node;
    node->next = curr;
    *head_addr = dummy.next;
}
```

This code allows us to insert values into their correct sorted position with ease, and have those changes reflected in the linked list itself. It makes the dummy node equal to the head of the list (not a copy of the list), and works forward from there.
A linked list that has the last node pointing at the first node is called a circular-linked list. It allows us to access everything from the position of tail instead.

Next, we will learn to search with sentinel. That is, it lets us traverse the list without checking if it is NULL. This is done by assigning an external node to a node, and advancing from there, repointing the new node at the next of it, and comparing the value.

A doubly linked list is a linked list that allows traversal in both directions.

$\emptyset$	$\leftarrow$	6	$\rightarrow$	$\leftarrow$	3	$\rightarrow$	$\leftarrow$	4	$\rightarrow$	$\emptyset$

It works almost entirely the same as a singly linked list, but has some more complications that arise when inserting or deleting nodes from the middle of the array. An example would be deleting, as shown here:

```
List-Delete(header, x):
    if x->prev != NULL
    (x->prev)->next = x->next
    else
    header->head = x->next
    if x->next != NULL
    (x->next)->prev = x->prev
    else
    header->tail = x->prev
```

As can be observed, the process involves a lot of back and forth, specially since we need to verify the existence of several nodes node. But, if it were a circular-linked list as well, the code could be reduced to:

```
List-Delete(x):
    (x->prev)->next = x->next
    (x->next)->prev = x->prev
```

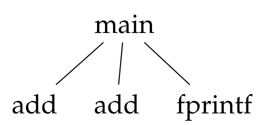
Recursion

Let us take a code snippet:

```
int add(int i, int j)
{
    return i+j;
}
int main(int argc, char **argv)
{
    int i, j;
    ...
    fprintf(stdout, "%d\n", add(add(i,j),j));
```

```
    ...
    return EXIT_SUCCESS;
}
```

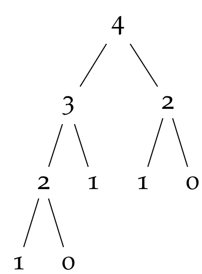
One might think that the first add is called by the fprintf function, however, that is not true. C needs to prepare all the inputs of a function before sending them into the function.
So, what actually happens is, main calls the inner add, takes that result, gives it to the outer add as a parameter, and finally takes that output and passes it to the fprintf function.
This process can be represented through a computation tree, which is, in fact, also very useful when analyzing recursive code.



Post-order traversal of these allow us to understand how these are used. So, in this case, the traversal will be:

$$\text{add} \rightarrow \text{add} \rightarrow \text{fprintf} \rightarrow \text{main}$$

Let us take recursive Fibonacci, for example:



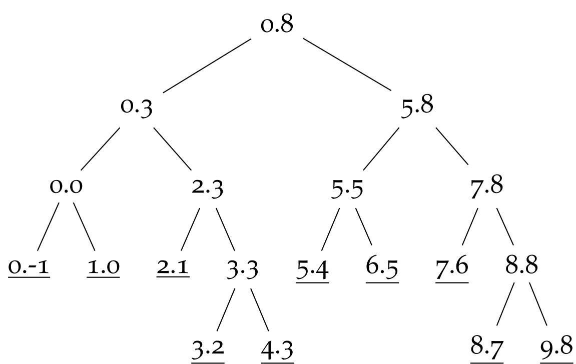
By conducting post-order traversal on this tree, we can see how the results will run, or in what order each value will be added.
The tallest stack in a recursive algorithm (or longest path in computational tree) is the space complexity. It is the additional memory necessary to store the data. This is because each call is allocating an additional constant amount of memory.
Note: if you were to have something like a[n] in your stack, the space complexity would drastically increase.
Let us look back at our Fibonacci tree once more. Let n be the number of sequence terms. Then T and L will be:

with T the total number of nodes, and L being the number of leaf nodes. So, $T = 1 + T(n-1) + T(n-2)$ and $L = L(n-1) + L(n-2) = Fibonacci(n)$.

n	T	L
0	1	1
1	1	1
2	3	2
3	5	3

So, if there are L leaf nodes, there will be L-1 non-leaf nodes, which is a total of 2L-1 nodes. And since L is F(n), we can calculate the complete space complexity using that.

A binary search tree for an array will have the following computation tree, for a sorted array of length 9:



if the code that defines it is:

```
Tnode *Build_BST(int *a, int lidx, int ridx)
{
    if (lidx > ridx)
        return NULL;
    int mid = (lidx + ridx)/2;
    Tnode *root = malloc(sizeof(*root));
    if (root != NULL) {
        root->data = a[mid];
        root->left = Build_BST(a, lidx, mid-1);
        root->right = Build_BST(a, mid+1, ridx);
    }
    return root;
}
```

This has a space complexity of O(log(n)). That is because at any given moment, there will be at most log(n) stacks (the height of the computational tree is O(log(n)). It will have a time complexity of O(n), as it is the number of non-leaf nodes in the computation tree. Similar trees can be built for any recursive algorithm.

"As an engineer, you must look for patterns. You won't see until you start looking."

- Prof. Koh

However, if there are nodes with non-constant time complexity, one cannot move forward under this assumption.
The recursive code snippet provided above uses two recursive calls. However, this can be reduced to a while loop.
That is, we can turn recursive functions into tail removal functions.
For example, the function:

```
function(a, b, c) {
    if (stopping_condition)
        base_case_body;
    other_divide_and_conquer_body;
    function(x, y, z);
}
```

Can instead be turned into:

```
function_tr(a, b, c) {
    while (not stopping_condition) {
        other_divide_and_conquer_body
        a = x, b = y, c = z;
    }
    base_case_body;
}
```

Now, let us look at the strategy for calculating time complexity of recursive calls.
We know that recursion takes a scenario, divides it into a sub-scenarios of size $\frac{n}{b}$, which it solves in the same way, and returns to combine the result of each of the sub-scenarios.
This means the time complexity of the full method will be: $T(n) = aT(\frac{n}{b}) + f(n)$ where $f(n)$ is the time complexity of the operations to combine the returned results. We also know that $T(1) = \Theta(1)$.
Let $c = \log_b(a)$.
This means that the number of leaf nodes will be n^c.
Since $T(1) = \Theta(1)$, leaf node computation will be $\Theta(n^c)$.
Now we will divide this into three cases:

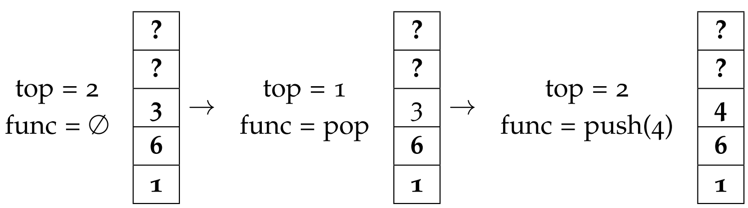
1. $f(n) = O(n^{c-\varepsilon}) \implies T(n) = \Theta(n^c)$
2. $f(n) = \Theta(n^c) \implies T(n) = \Theta(n^c \log(n))$
   More generally, if $f(n) = \Theta(n^c \log^k(n))$:
   - $k > -1 \implies T(n) = \Theta(n^c \log^{k+1}(n))$
   - $k = -1 \implies T(n) = \Theta(n^c \log(\log(n)))$
   - $k < -1 \implies T(n) = \Theta(n^c)$
3. $f(n) = \Omega(n^{c+\varepsilon}) \implies \Theta(f(n))$

Stacks

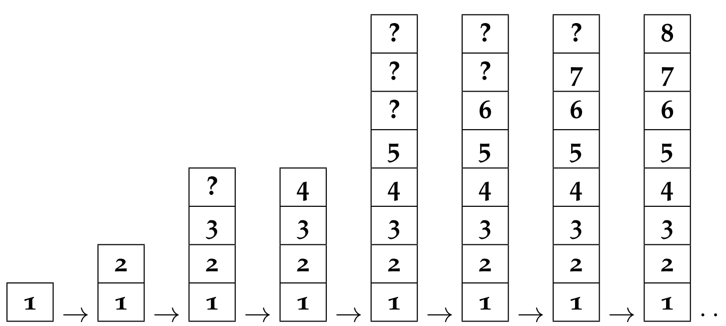
Stack: Collection of items of which only the top item can be accessed.
Works on the Last In First Out (LIFO) rule.
It has four primitive operations, all of which have $O(1)$ time complexity:

- Push(S,i): adds item to top of stack
- Pop(S): remove top item from stack and return it
- Stacktop(S): return top stack item
- Empty(S): returns whether the stack is empty

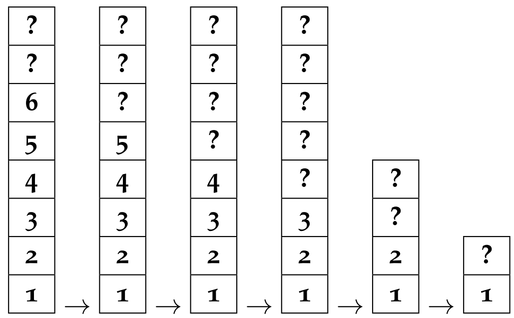
A stack can be simulated as a linked list, with the head of the linked list being the top of the stack.
Another method is to use an array, with a struct defining the address of the array, the size of the array, and the index of the stack top.
It behaves as the following:



A stack should be grown if the number of elements to be stored in the stack is greater than the memory available. A good way to do so is to double the size each time the limit is reached, allowing for allocation of several new terms without allocating too large or too small an amount of memory.



On the other hand, one should shrink only when it has gone an exponent below the limit, that is, if we had expanded to 5, and shrink to 4, we should not shrink it instantly, we should instead wait until it has reached to two. This will allow us to not have to shrink and expand repeatedly if we push and pop many times in a row around the transition value.



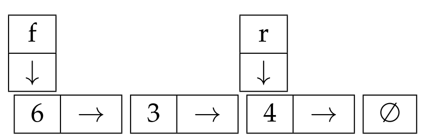
Queues

Queues, unlike stacks, which work on the LIFO rule, work on the FIFO rule. That is, you can add only at the end, but pop or remove and access at the beginning.

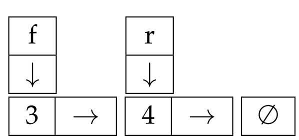
The primitive operators are:

- enqueue(Q,i): add item at end of queues
- dequeue(Q): remove and return item from front of queue
- Front(Q): return front of queue
- Rear(Q): return end of queue
- Empty(Q): return whether or not the queue is empty

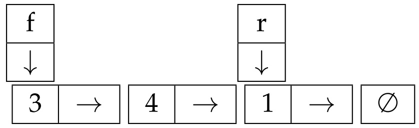
Once again, these can be represented through linked lists. The primitive operations will have $O(1)$ complexity since the tail and head are previously stored.



If we were to dequeue:



If were to enqueue(1) from there:



This can also be implemented with arrays.
We will store the address of the array, the size of the array, the front and the rear of the array.

1	2	3	?

f=0, r=3

1	2	3	4

f=0, r=4

1	2	3	4

f=1, r=4

1	2	3	4

f=2, r=4

5	2	3	4

f=2, r=1

5	6	3	4

f=2, r=2

How to enqueue now?

Option 1:

5	6	3	4	?	?	3	4

f=6, r=2

Option 2:

5	6	3	4	5	6	?	?

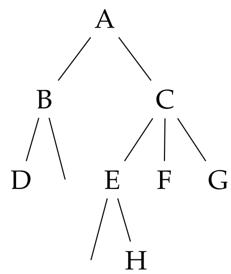
f=2, r=6

Continue as before...

Queues can also be double ended. These are known as dequeues. They are implemented similarly, however, the linked list is changed for a doubly linked list.
There also exist priority queues, in which items are added or removed in order of priority. For these, the time complexities to enqueue and dequeue are O(1) and O(n) respectively. However, if it were ordered, it would be O(n) and O(1).

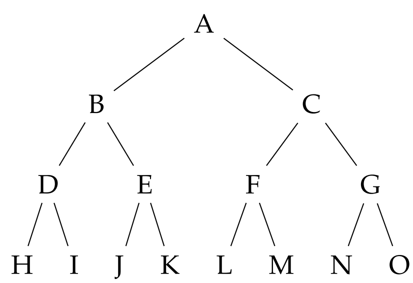
Trees

A tree is a graph such that there exist no cycles between its nodes. Have an algorithmic reduction in time complexity of searching and sorting elements.

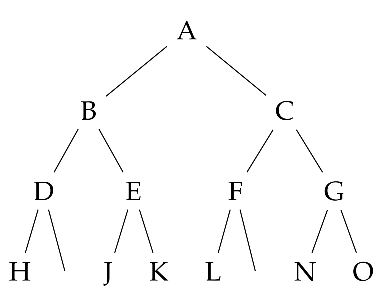


In which A is a root node.
B and C are sibling nodes, as they are on the same level.
E is a child of C, and C is a parent of E.
F is a descendant of A.
D and H are both leaf nodes.
A is on level 0, B and C are on level 1, etc.

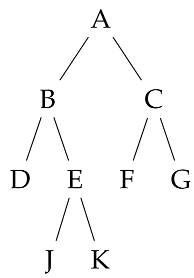
Had there been more than one method to get from any node to another, this would not be a list.
A complete binary tree is that which has all levels completely filled:



At level L, there will be 2^L nodes, the total number of nodes, in terms of the height h will be $2^{h+1} - 1$. This means the height will be O(log(n)).
An almost complete binary tree is one that is full on all levels except the last:



Will have at least 2^h nodes, and at most $2^{h+1} - 2$. A strictly binary tree is one in which each node has either 0 or 2 nodes.



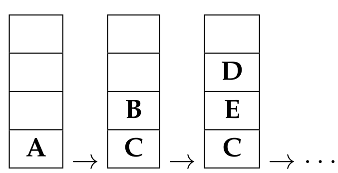
If there are n leaf nodes, there will be 2n-1 non-leaf nodes.
It is defined as following:

```
typedef struct _Tnode
{
    int data;
    struct _Tnode *left;
    struct _Tnode *right;
} Tnode;
```

This can in fact also be implemented with arrays.
There are four kinds of list tree traversal:

1. Preorder: root, left, right
2. Inorder: left, root, right
3. Postorder: left, right, root
4. Breadth first: by level, right to left.

A tree can also be traversed through by creating a stack:



With each pop from a stack printing and adding the two children into the stack.
Alongside that, a queue can be used to conduct breadth traversal.

"Fixes"

Prefix, postfix, and infix notations are those which come from preorder, postorder, and inorder traversals of the following tree:

```
          -
         / \
        /   *
       / \ / \
      +  c d  e
     / \
    a   b
```

preorder	inorder	postorder
-(/(+(ab)c)*(de))	((a+b)/c)-(d*e)	ab+c/de*-

These expressions allow a computer to easily evaluate mathematical inputs it receives.

BSTs and AVLs

Binary Search Trees

Now that we know what trees are, we will define Binary Search Trees, or BSTs.
A BST is a tree such that all descendants of a node to the left of the node are less than the node, while all nodes to the right are larger than the node.
This leads to inorder traversal of the tree being sorted.
To search for a number within a BST, one can simply use the following code snippet:

```
Tnode *binary_search(Tnode *root, int key)
{
    Tnode *curr = root;
    while (curr != NULL) {
        if (key == curr->data)
            break;
        else if (key < curr->data)
            curr = curr->left;
        else
            curr = curr->right;
    }
    return curr;
}
```

This code snippet basically goes right if the number being searched is larger, and left otherwise.
The problem with binary search is that, in the worst case scenario, which is when the tree is completely skewed, the time complexity

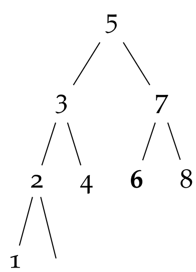
is O(n), as opposed to what a binary tree should be able to provide, which is O(log(n)).

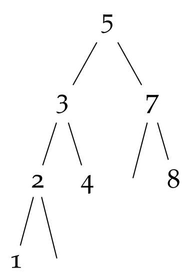
The direct method of inserting a node into a binary search tree entails the following.

- Traverse the tree as if searching for the number to be inserted.
- When you reach an "empty space", or a NULL node, create a node and place the number at the spot.

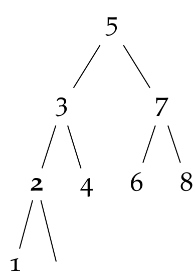
The worst case time complexity of this is once again O(n).
Node deletion is slightly more tricky. It can be divided into three cases.

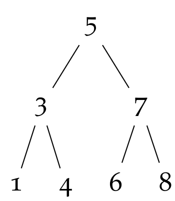
1. No children: Search for the node, if it exists, free memory and all links to the node.



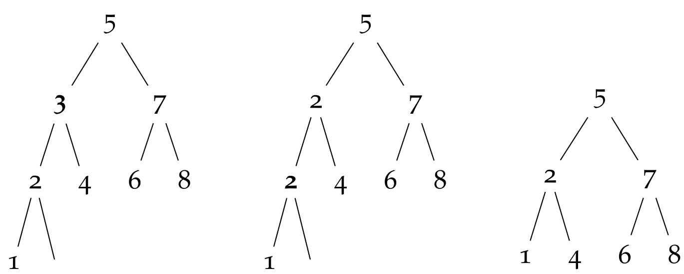


2. One child: Search for the node. Make parent of node to be deleted, parent of child of node to be deleted. Free memory.



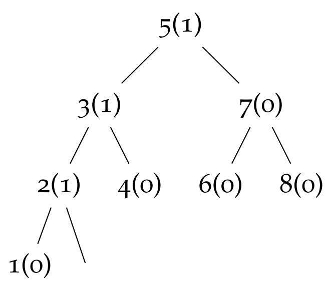


3. Two children: if a node has two children, we replace the key of the node to be deleted with that of its immediate inorder predecessor (That is, go left once, and then right all the way.). Then delete the inorder predecessor.



Height Balanced BSTs (AVLs)

As mentioned in the previous section, the time complexity for BST traversal, insertion, and deletion can reach up to O(n). This is highly inefficient. To remove these inefficiencies, the AVL was created.
An AVL is a BST such that the difference in height between the right branch and left branch of any node, is at most one.
To maintain the balance of a tree, we undergo steps known as rotations.
Moving forward, in each node, we will store the key, the children and the balance, where the balance is defined as the height of the left branch minus the height of the right branch.



We want the balance of all nodes to be 1,0 or -1 at the end of any insertion or deletion operation.
We will begin by understanding how to insert a node.
Node insertion requires a few steps after the steps mentioned before. So, let us assume that a node has been successfully inserted into the tree. By moving up the tree recursively, we can update balances based on which of the two children the recursive call addressed. That is, if the child that had its balance updated was the right child, if the parent were to require a balance update, it would change from 1 to 0, 0 to -1, or -1 to -2.
Let us look at how balances are updated:

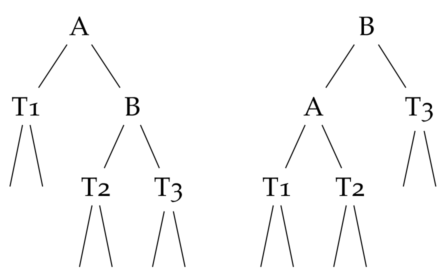
- If the initial balance was 0, and a node was inserted, it would become 1 or -1, and the parent would need to have its balance updates as well.

- If the initial balance was -1 or 1, and after the node was inserted, it became 0, the height of the tree remains the same, and thus, the balance of the parent does not need to be updated.
- If the initial balance was -1 or 1, and after the node was inserted, it became 2 or -2, rotations are necessary, but parent will no longer require balance updates.

We will now define rotations. There are two kinds of rotations, clockwise rotations (CR), and counter-clockwise rotations (CCR). The following is the code definition of CCR:

```
CCR(old_root) {
    new_root = old_root->right;
    old_root->right = new_root->left;
    new_root->left = old_root;
    return new_root;
}
```

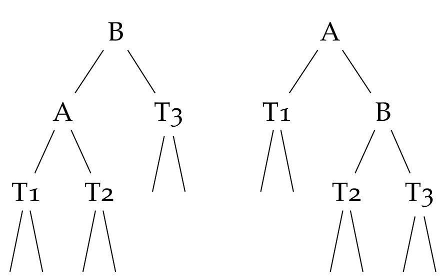
What this code does is transform a tree as follows:



On the other hand, the following is a definition of CR:

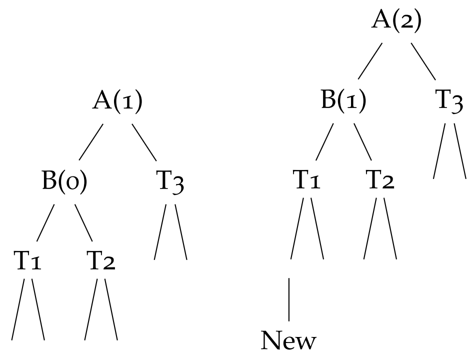
```
CR(old_root) {
    new_root = old_root->left;
    old_root->left = new_root->right;
    new_root->right = old_root;
    return new_root;
}
```

Which does:

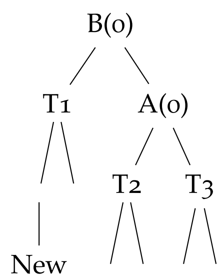


So, how are these relevant to balancing a tree?

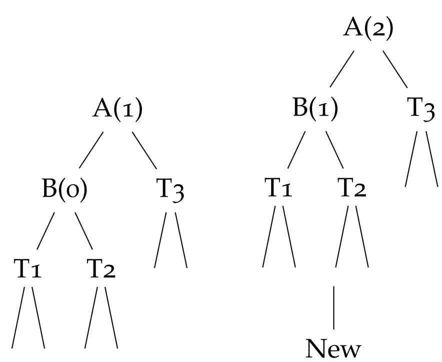
- Case 1:



By applying CR(A):

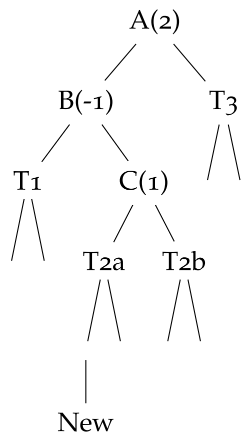


- Case 2:

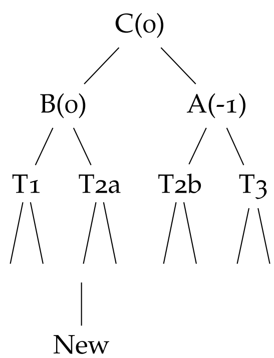


This needs two rotations to balance, first, CCR(B), then CR(A). The end result of this can be subdivided into further cases:

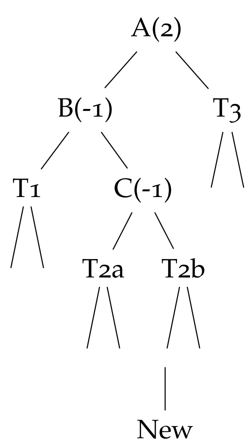
  - Case 2a:



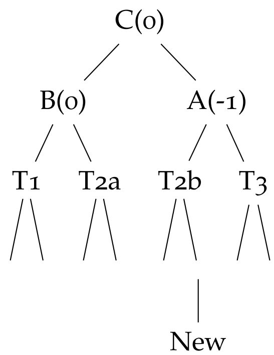
Which, after the two operations ends up as follows:



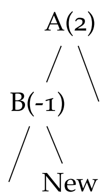
– Case 2b:



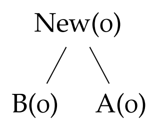
Which, after the two aforementioned operations, ends up as:



- Case 2c:



which will end up as:



- Case 3: Mirror of Case 1
- Case 4: Mirror of Case 2

Now we will learn how to delete.
The deletion process is like the mirror of the insertion process.
The differences are reflected in things like, if there was a node deleted in the right root, the balance changes from 0 to 1.
If the balance changes from 0 to anything, there is no change in height.
However, if the balance changes from -1 to 0, or 1 to 0, there is a change in height that will be reflected in the parent.
Finally, if the balance changes from -1 to 1 or vice versa, there was no change in height.
The time complexity of both the deletion and insertion is O(log(n)).

Heapsort and Priority Queues

Heapsort

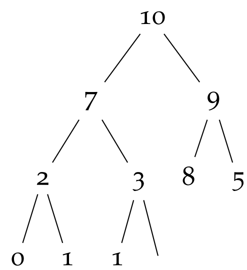
A heapsort is a tree represented by an array. It is also called a max-heap. It is an almost complete binary tree such that the value of all children is less than or equal to the value of the node itself.
It has O(log(n)) insertion and deletion operations.
Its representation as an array is the following:
Root has index 0.
Parent of (i>0) has index value in array $\frac{i-1}{2}$.

Left of (i) has index of $2i + 1$ as long as the index is within array bounds.

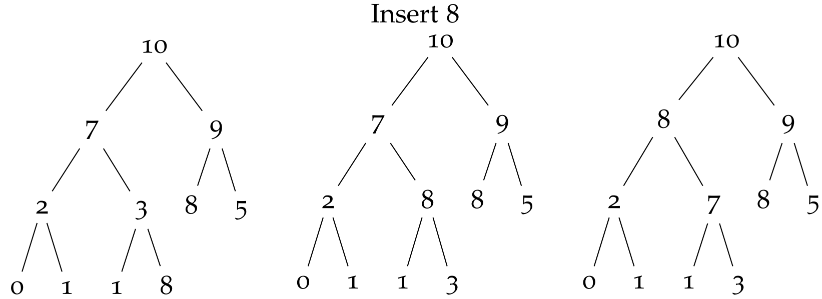
Right of (i) has index $2i + 2$.

Steps to add node:

1. Insert node at end of heap.
2. If element is greater than parent, exchange position with parent.



Insert 8



The following is the code for this process:

```
Upward_heapify(array[], n):
    new = array[n]
    child = n
    parent = (child-1)/2
    while child > 0 and array[parent] < new
        array[child] = array[parent]
        child = parent
        parent = (child-1)/2
    array[child] = new
```

The dequeue operation entails the following steps:

1. Exchange root and last node
2. Exchange new root with larger of two children while any of the two children is larger.

The code for this is:

```
Downward_heapify(array[], i, n):
    temp = array[i];
```

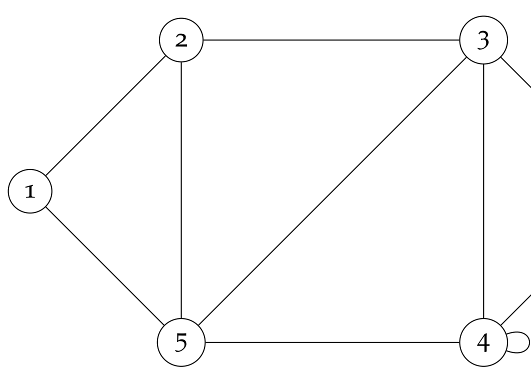
```
while ((j = 2*i+1) <= n)
    if (j < n and array[j] < array[j+1])
        j = j+1
    if (temp >= array[j])
        break
    else
        array[i] = array[j]
        i = j
array[i] = temp
```

One can sort by enqueueing all elements and then repeatedly dequeue the root.
One can more efficiently build the tree using downward heapify using the following code:

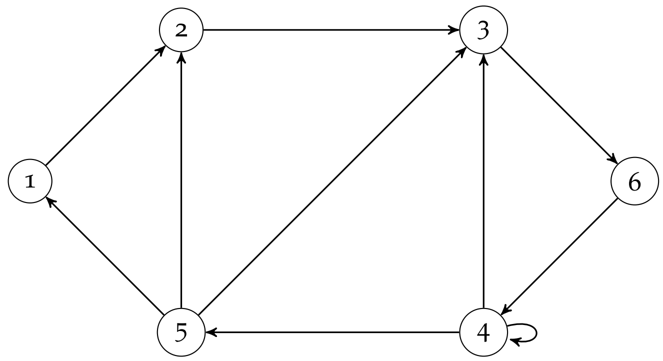
```
for i = n/2 - 1 downto 0
    Downward_heapify(r[], i, n-1)
for i = n-1 downto 1
    swap(r[i], r[0])
    Downward_heapify(r[], 0, i-1)
```

Graphs

A graph consists of a set of vertices and edges. Each edge connects two nodes.



On the other hand, a directed graph is that in which each edge has a direction.



This leads to a few definitions:

- For a directed edge <u,v>:
  - The edge is incident from u and incident to v
  - v is adjacent to u
  - u is the predecessor of v
  - v is the successor of u
  - u is the source node
  - v is the destination node
- For an undirected egde (u,v):
  - the edge is incident on both nodes
  - both nodes are adjacent to the other
- Degree: number if incident edges for a node in an undirected graph
- Indegree: number of incoming edges for a node in directed graph
- Outdegree: number of outgoing edges for a node in directed graph

A weighted graph is that which associates a number with each edge, known as the weight.
A path of length k is a sequence of $k+1$ adjacent nodes.
A path from a node to itself is a cycle.
If a graph has no cycles, it is acyclic, otherwise, it is cyclic.
A graph is connected if every node is reachable from every other.
A directed graph is strongly connected if every node is reachable from every other.
Note: a tree is an acyclic graph.
A graph can be represented in two main ways:

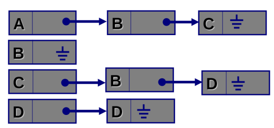
- Adjacency-matrix representation: it is a $V \times V$ matrix such that it has True in every node that is adjacent to it.

to

from	A	B	C	D
A	F	T	T	F
B	F	F	F	F
C	F	T	F	T
D	F	F	F	T

If it were weighted, it stores the weight instead. For the empty spots, they can be filled with 0 or ∞ as required by the problem to be solved.
Has high memory requirements.

- Adjacency-list representation: It is a list that stores each node. Each node has a list which stores all the adjacent nodes.



Each of these representations has their own advantages and disadvantages depending on what needs to be done: There are several

Comparison	Winner
(x,y) exists?	matrix
find vertex degree	list
less memory for small	m+n vs n^2
less memory for big	matrices
edge insertion	matrices
traversal	m+n vs n^2
most problems	lists

ways to traverse graphs. We will now cover two of them, which are Breadth-First Search (BFS), and Depth-First Search (DFS).

- BFS: Explore all nodes at a distance k before exploring any at distance k+1.
  The unexplored nodes are stored via queues.
  We will use three colors to represent the status of a node:
  - White: Never been visited.
  - Gray: Has been visited, but still in queue.
  - Black: Fully analyzed, no longer in queue.

  The code that conducts BFS is the following:

```
BFS(G)
// G: adj-list representation
for each node u in V[G]
    color[u] = WHITE
    d[u] = infinity
    parent[u] = NULL
for each node u in V[G]
    if (color[u] == WHITE)
        bfs(G, u)
```

and the function bfs is:

```
bfs(G, s)
    Q =init_queue()
    color[s] = GRAY;
    d[s] = 0; parent[s] = NULL
    Enqueue(Q, s) // insert into Queue
    while (!Empty(Q))
        u = Front(Q)
        for each v in adj[u]
            if (color[v] == WHITE)
                color[v] = GRAY;
                d[v] = d[u] + 1; parent[v] = u
                Enqueue(Q, v) // insert into Queue
    Dequeue(Q)
    color[u] = BLACK
```

In essence, what BFS does is the following:

1. Visit first node
2. Visit all adjacent nodes of node as long as they are white (unvisited)
3. Remove first node from queue
4. Go to next node in queue
5. Repeat from step 2 until graph completely traversed

This leads to a few definitions:

- Tree edges: any edge of the BFS tree(s)
- Back edges: edges connecting a vertex to an ancestor, such as self loops
- Cross edges: all other edges

BFS has a time compexity of O(V+E)

- DFS: explore all reachable nodes, before changing origin.
  Once again, we will use the same colors with the same representations.
  The code is the following:

```
DFS(G)
    // G: adj-list representation
    for each node u in V[G]
        color[u] = WHITE
        parent[u] = NULL
    time = 0
    for each node u in V[G]
        if (color[u] == WHITE)
            dfs(G, u)
```

  where dfs is:

```
    dfs(G, u)
    color[u] = GRAY // visit begins
    d[u] = time = time + 1
    for each v in adj[u]
        if (color[v] == WHITE))
            parent[v] = u
            dfs(G, v)
    color[u]= BLACK // visit ends
    f[u] = time = time + 1
```

  In essence, what DFS does is the following:

  1. Visit first node
  2. Go to top priority node from there (recursively)
  3. Repeat this process until no visitable nodes that are unmarked exist
  4. Go back to step 1 but choose new, unvisited node
  5. Repeat until graph has been completely visited

  Within the DFS trees and forests, there exist the following definitions:

  – Tree edge: any edge from the tree(s)
  – Back edge: edges connecting to an ancestor in a DFT
  – Forward edges: nontree edges connecting a vertex to a descendant
  – Cross edges: all other edges

Shortest Path Algorithms

A shortest path has the following properties:

- Any subpath of a shortest path is a shortest path
- A path δ fulfills: $\delta(s, v) = \delta(s, u) + w < u, v >$