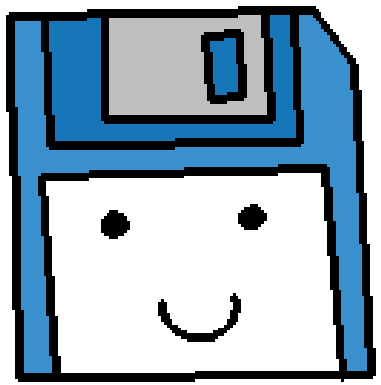
Introduction to Searching and Sorting

Written by Rachel J. Morris, last updated 2016-04-10

Searching for Data



Searching

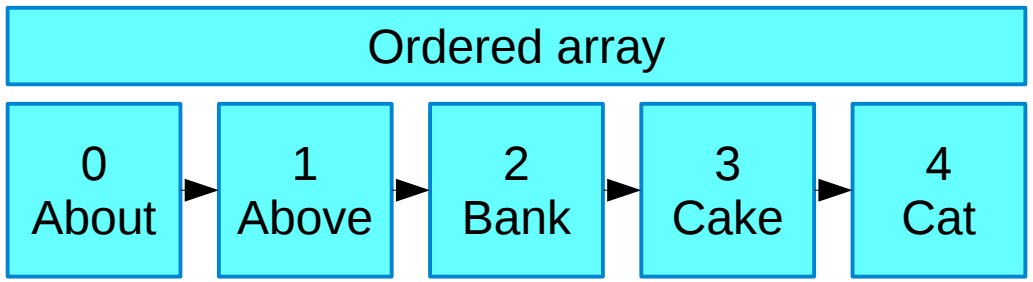
Searching is one of those tasks that is tedious and slow for a human, but can be automated by a computer.

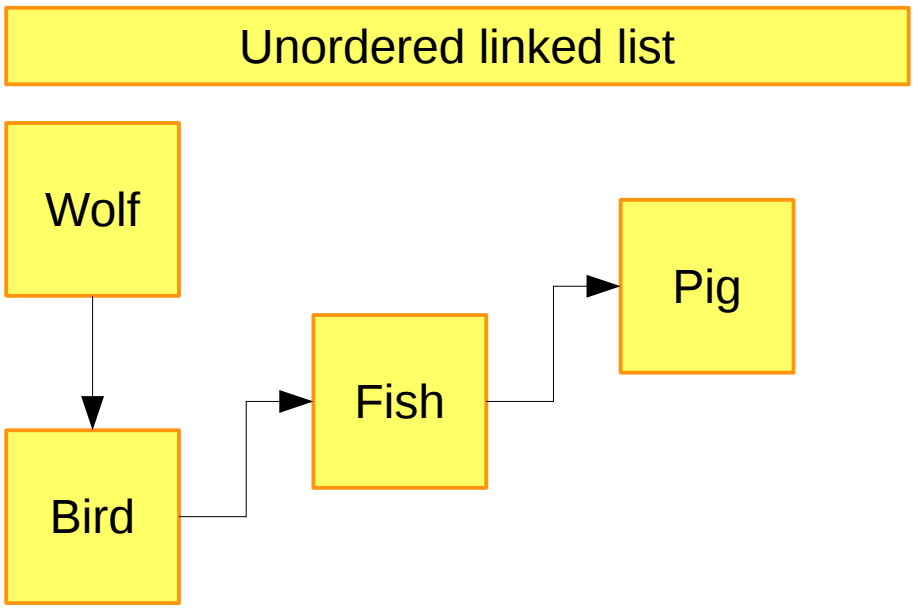
When we're storing so much data in a database or other format, we need a way to access that data by some criteria.

Searching

There are different algorithms for **searching**.

There may be different ways to approach searching, based on the type of data structure that you are searching, and whether the data in that structure is sorted or not.





Linear Search

```
int FindName( const string names[], int listSize, const string& searchTerm )
{
    for ( int i = 0; i < listSize; i++ )
    {
        if ( names[i] == searchTerm )
        {
            return i;
        }
    }
    return -1; // not found
}
```

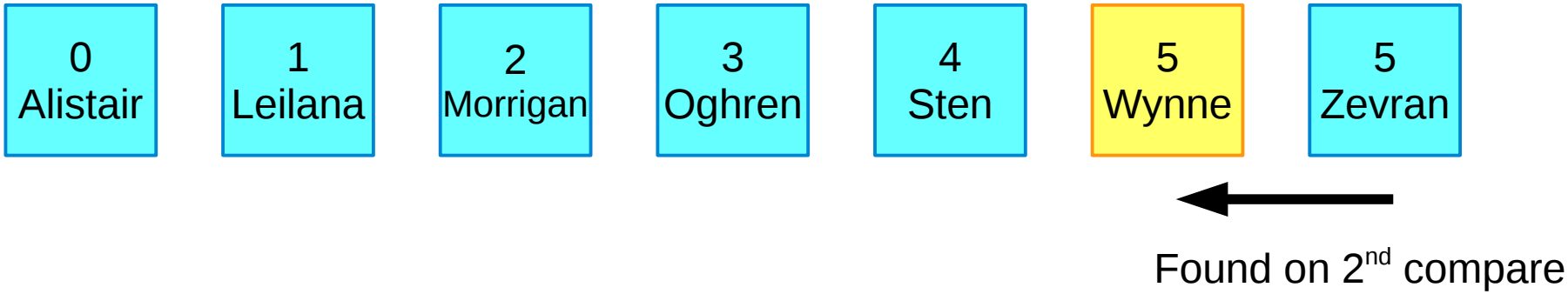
The easiest to implement is a search from the beginning to the end of a list, checking each item in order.

This search is pretty “dumb”. How could it be improved?

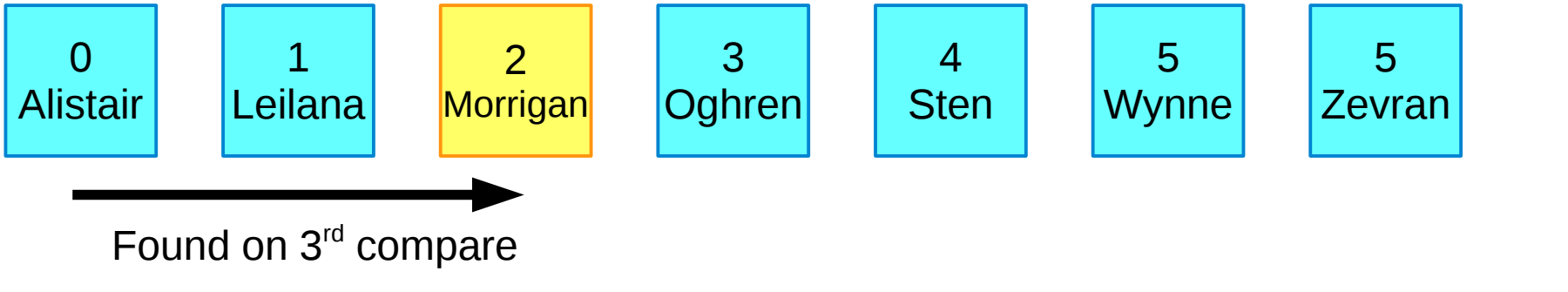
Thinking of strategies...

If the list is in alphabetical order, we could check the first letter, and decide to begin searching from first → last, or last → first...

`Find "Wynne"`

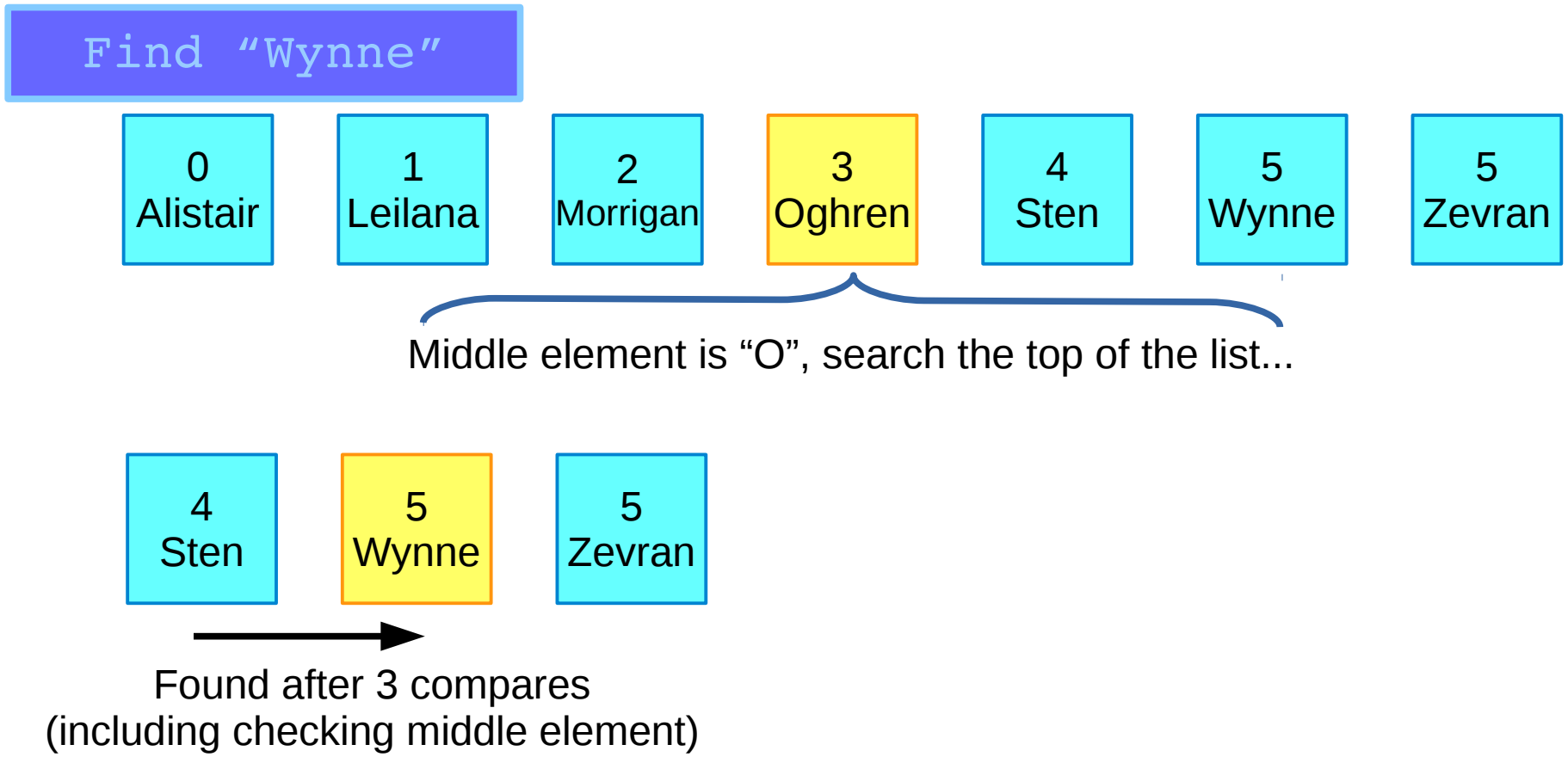


`Find "Morrigan"`



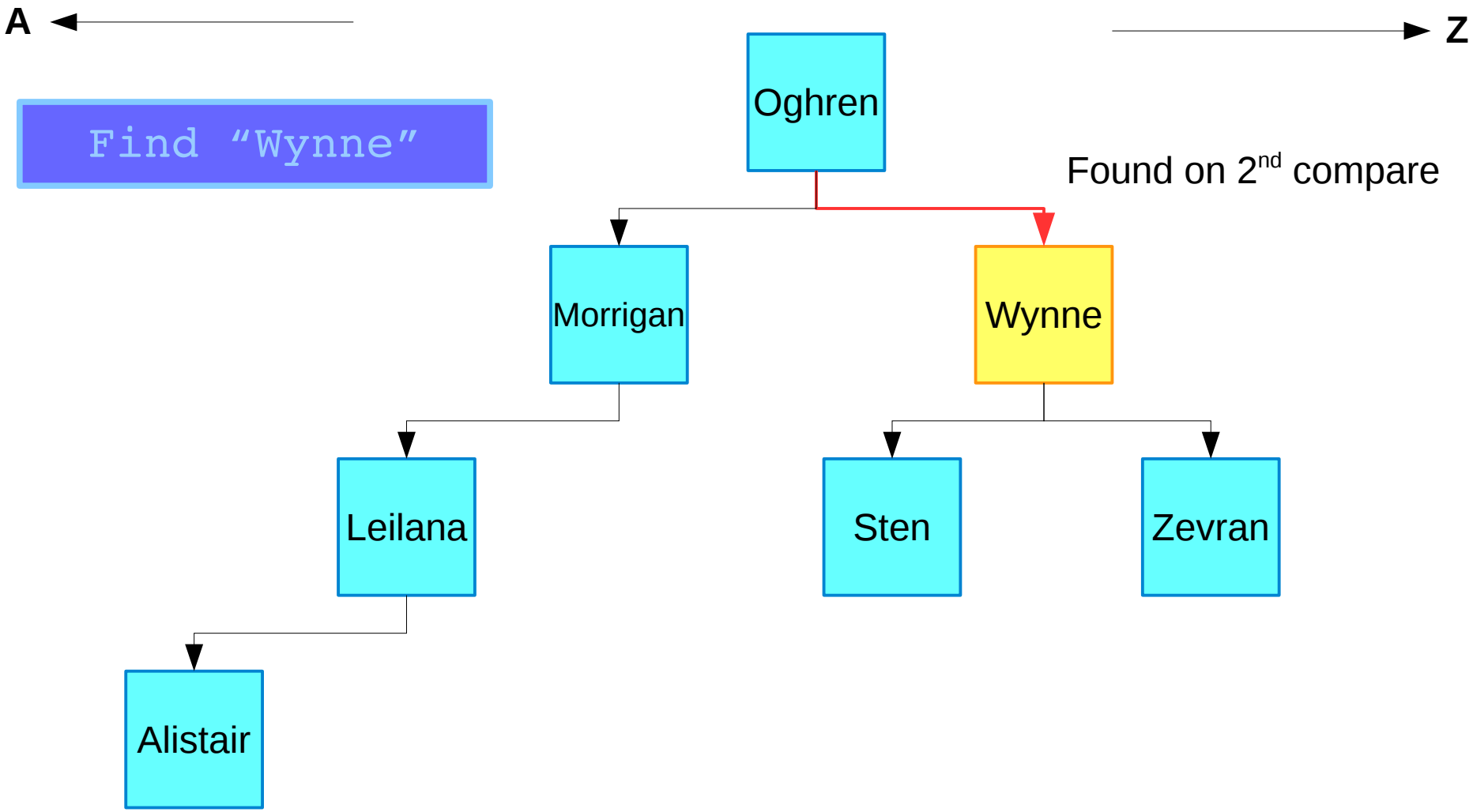
Thinking of strategies...

If the list is in alphabetical order, we could check the middle element, and figure out which half of the list to search based on alphabetical order...

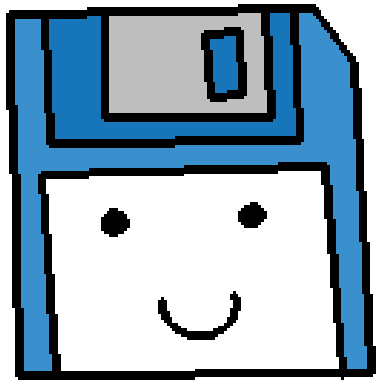


Thinking of strategies...

We could store the “names” list in a different data structure to make it easier to search for items without covering the entire list...



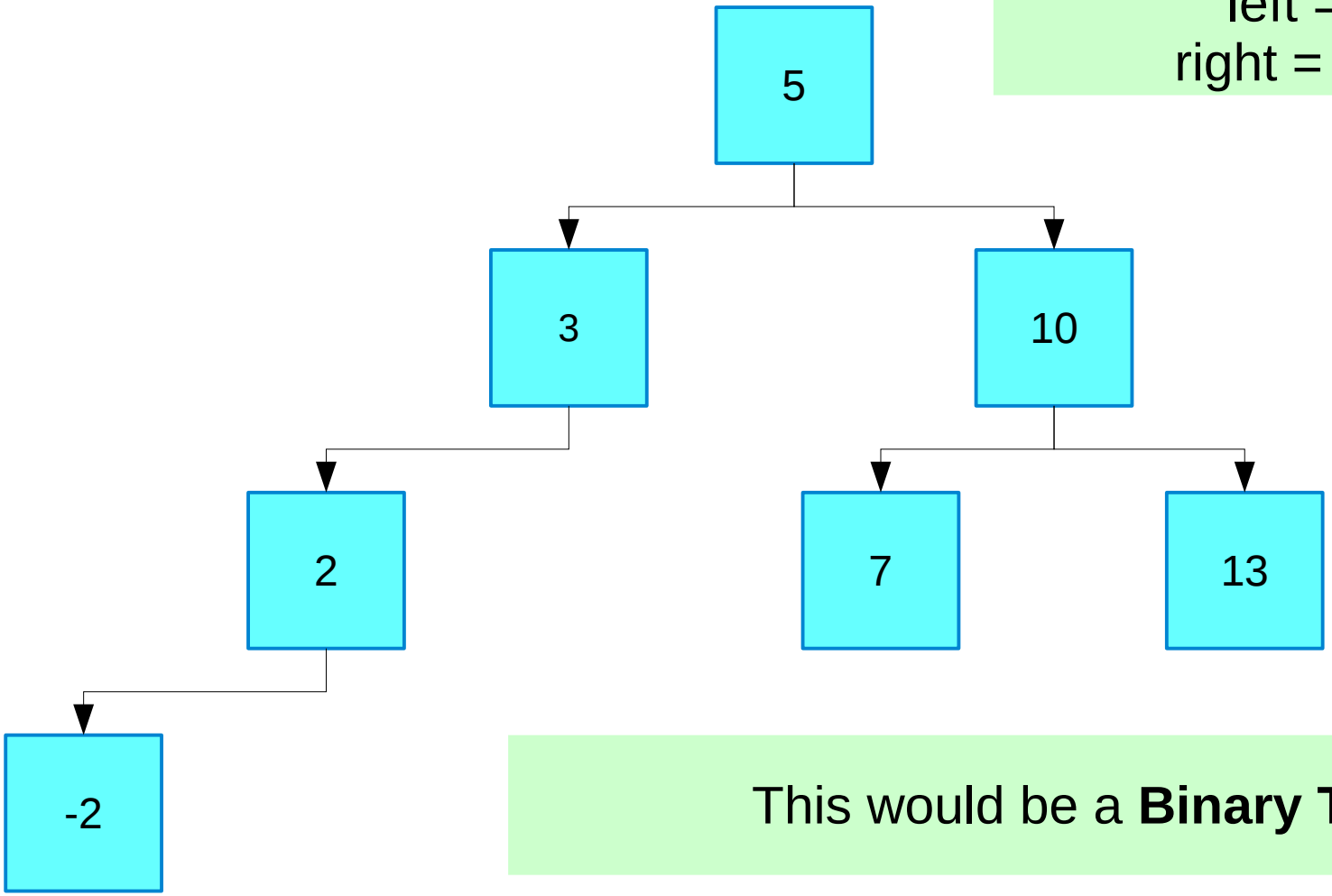
Introduction to Binary Trees



Intro to Trees

What if you want to store your data like this?

Notice that everything is sorted so that
left = less
right = greater



This would be a **Binary Tree**

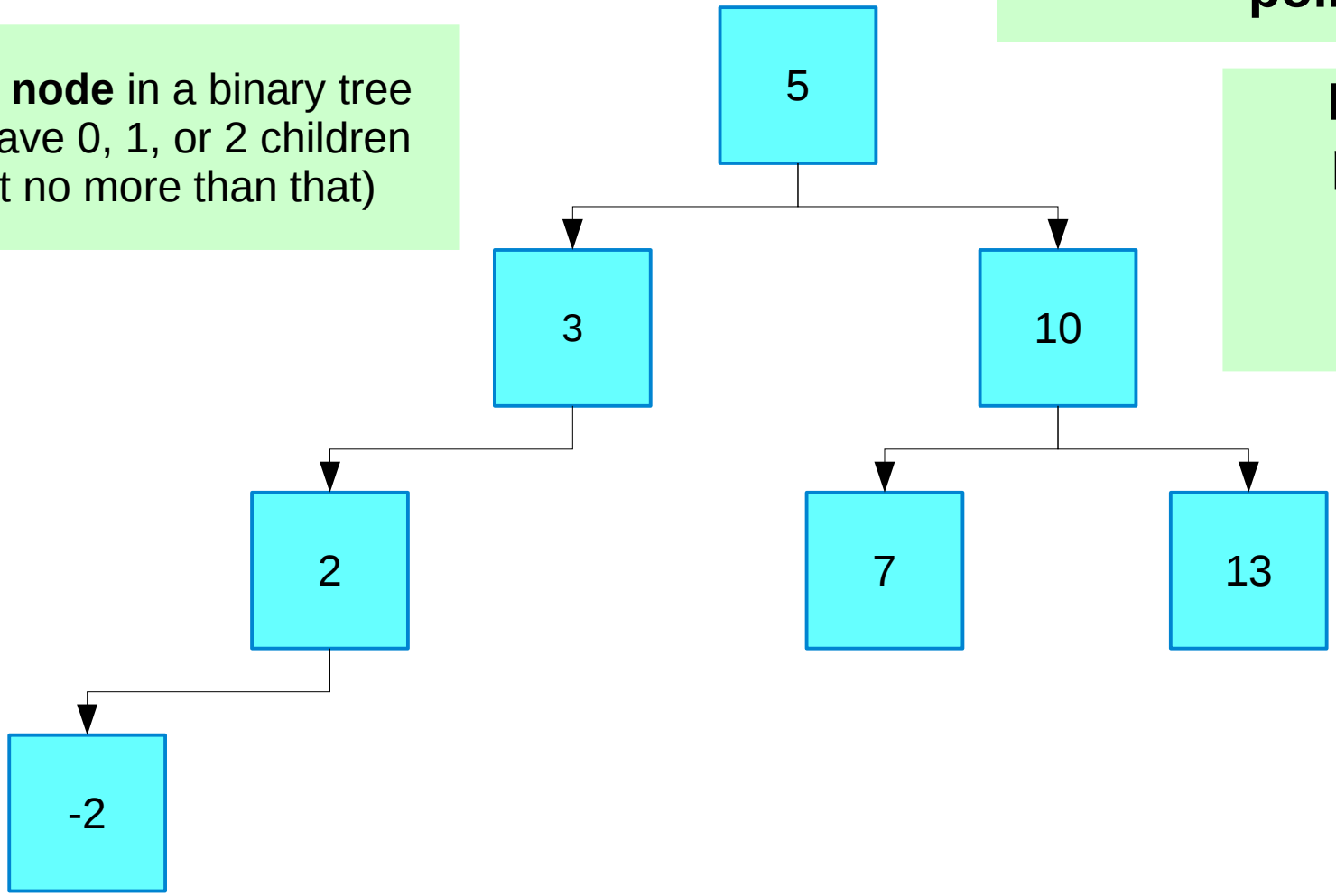
Intro to Trees

For the binary tree structure, there is a collection of **nodes**.

We could implement a data structure like this utilizing **pointers**

Each **node** in a binary tree can have 0, 1, or 2 children (but no more than that)

Each **node** points to its **left-node** & **right-node**

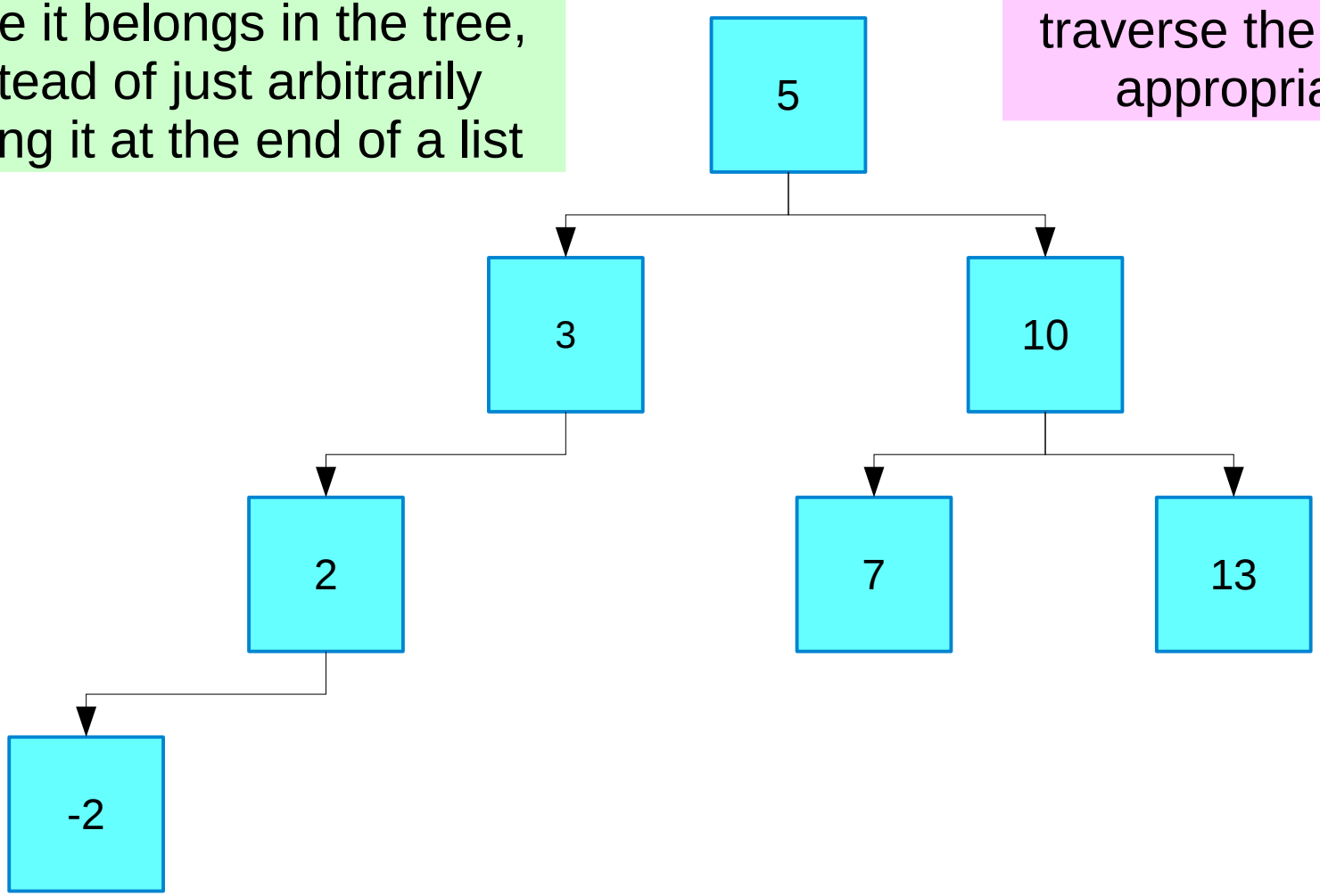


Intro to Trees

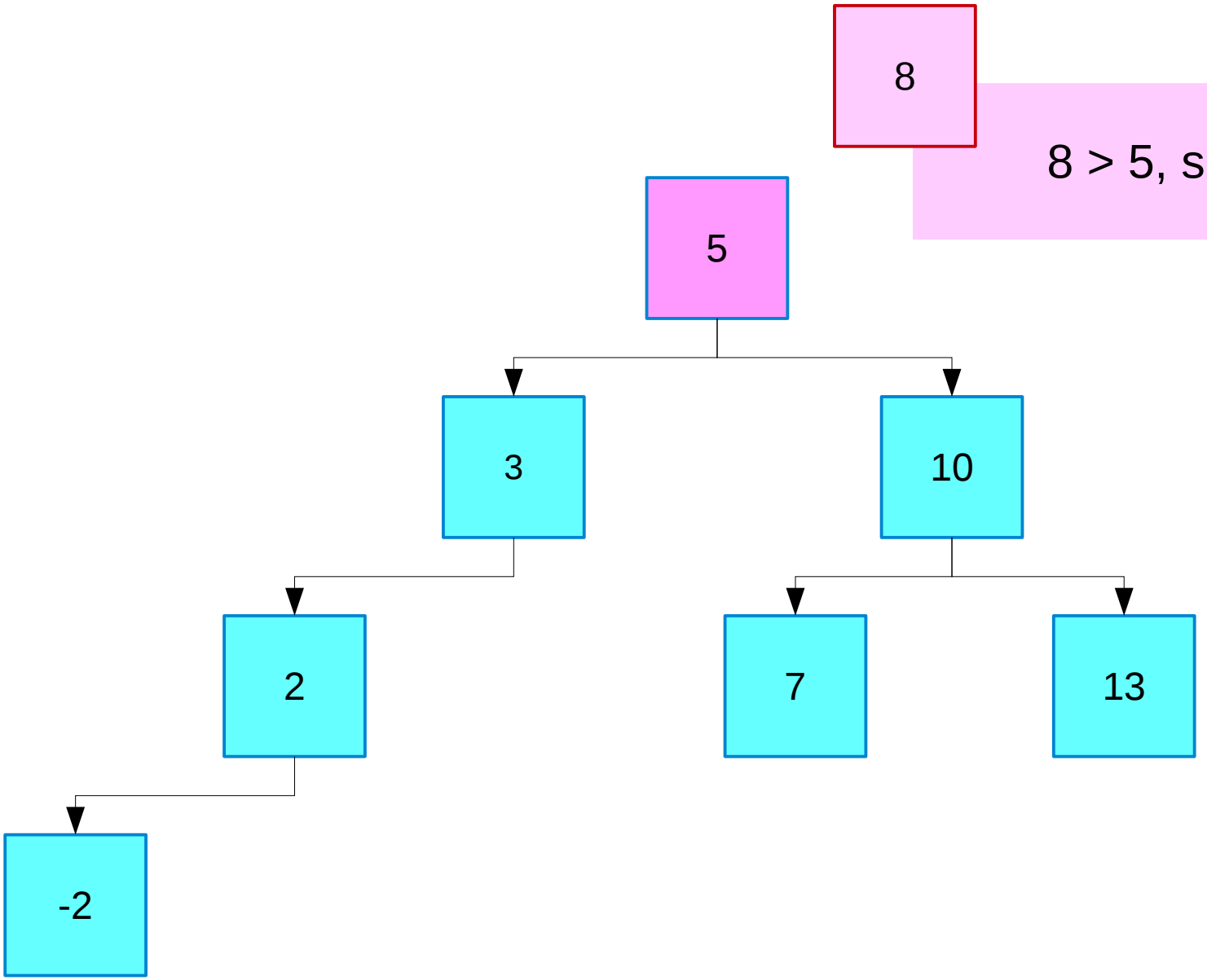
When inserting a new element, we can keep the tree sorted by looking for where it belongs in the tree, instead of just arbitrarily placing it at the end of a list

8

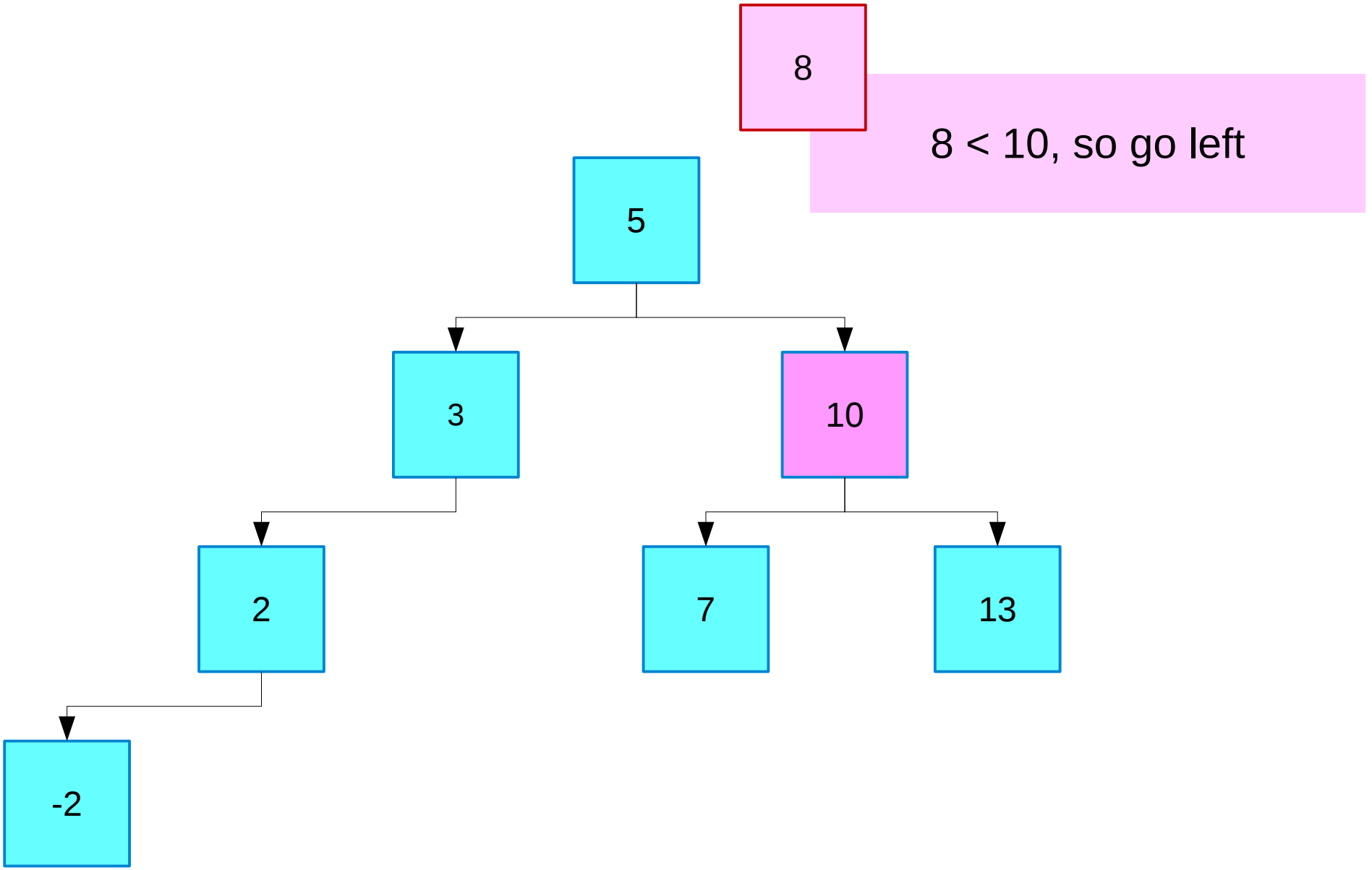
When we insert a new element, we need to traverse the tree to find an appropriate position



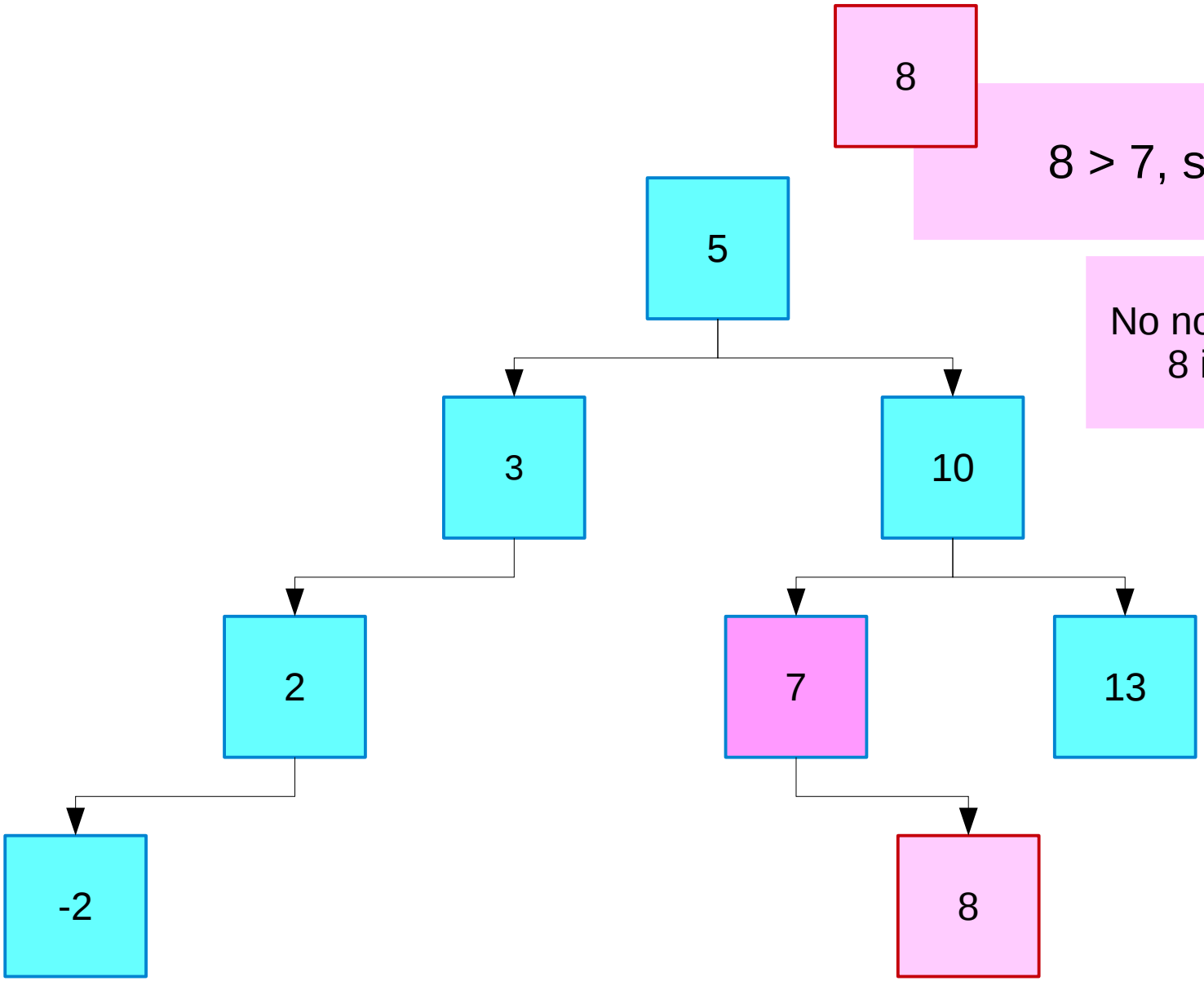
Intro to Trees



Intro to Trees



Intro to Trees



8 > 7, so go right

No node here, so insert 8 in this position.

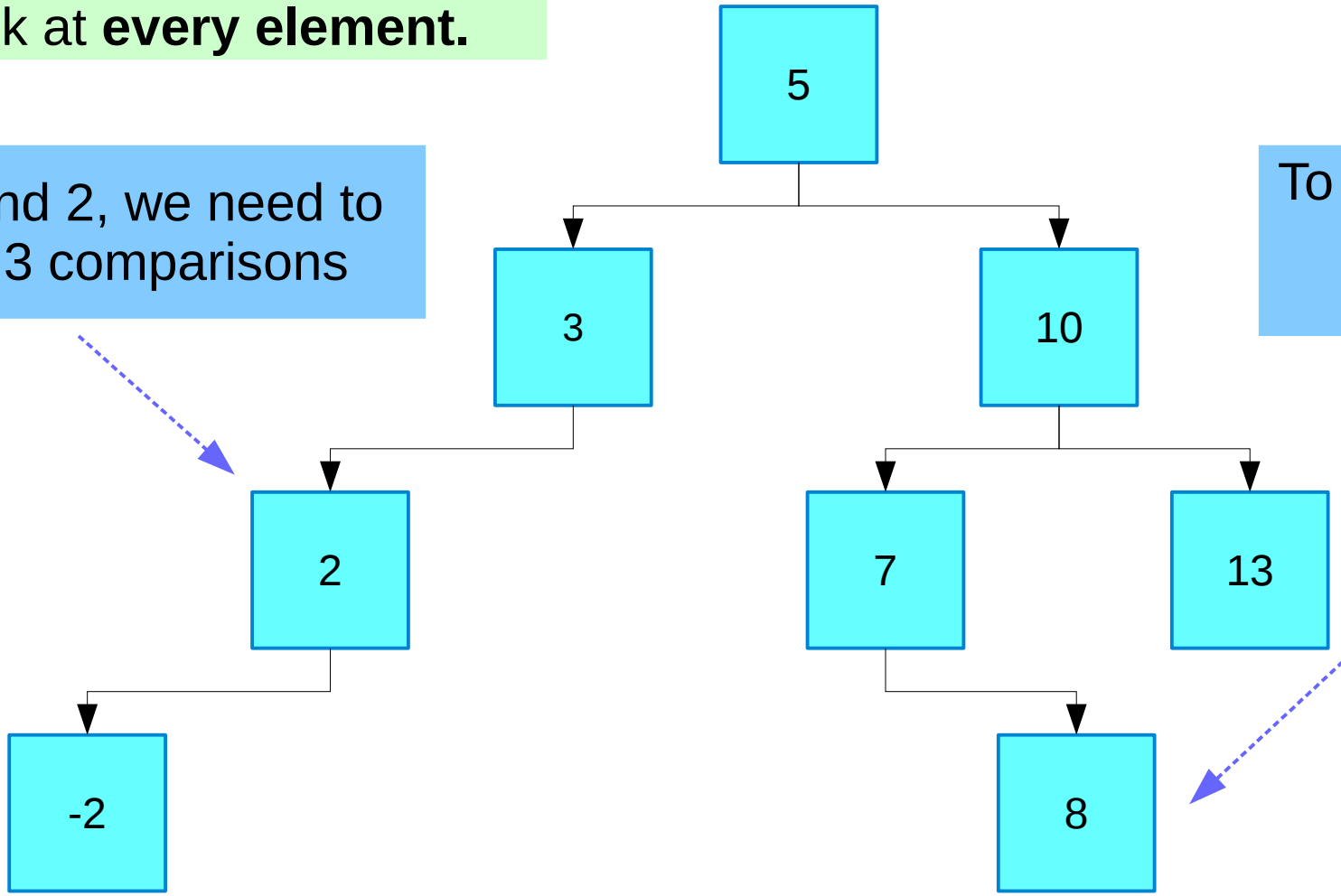
Intro to Trees

The binary tree is still in order, and we can find values without having to look at **every element.**

No matter what we're looking for, the amount of comparisons we do is **less than the size of the tree.**

To find 2, we need to do 3 comparisons

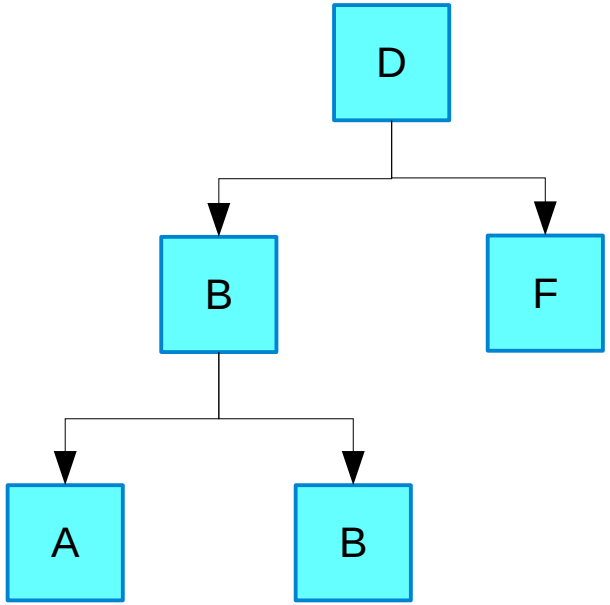
To find 8, we need to do 4 comparisons



Intro to Trees

So what would a binary tree look like in code?

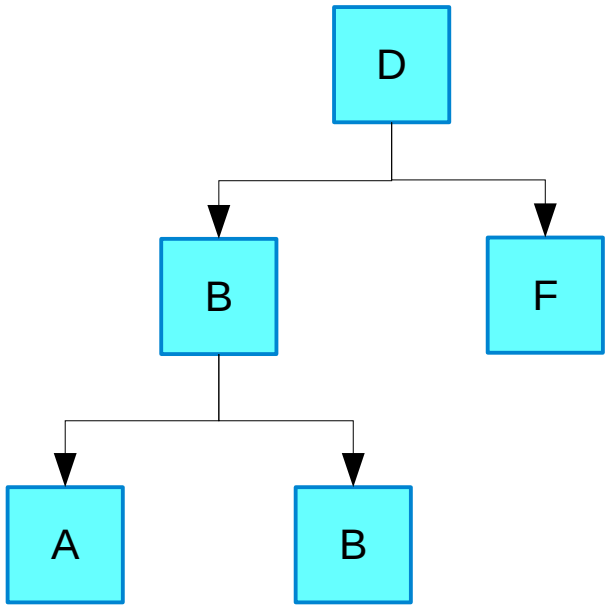
We need a **node** object, and a **binary tree** object.



For now, we are going to stick to storing one type of data (a char), but later on we can write code with **templates**, that will allow us to write one structure to store any type of data.

Intro to Trees

```
class Node
{
    public:
    Node()
    {
        left = NULL;
        right = NULL;
    }

    char data;
    Node* left;
    Node* right;
};
```



A node will store some data and keep track of its neighbors.

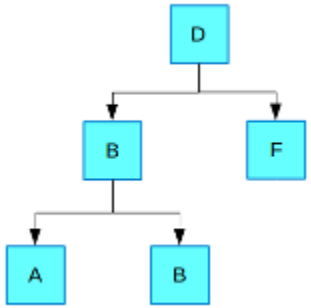
Generally, the left neighbor is **less** and the right neighbor is **greater**

Intro to Trees

```
class Node
{
    public:
    Node()
    {
        left = NULL;
        right = NULL;
    }

    char data;
    Node* left;
    Node* right;
};
```

```
class BinaryTree
{
    public:
    BinaryTree();
    ~BinaryTree();

    void Insert( char data );
    Node* GetNode( char data );

    private:
    Node* CreateNode( char data );
    void DestroyTree();

    Node* m_root;
    int m_size;
};
```

Our tree object will store a pointer to the **first node: The root**



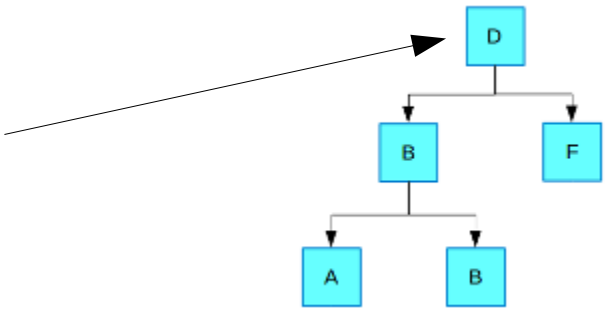
Intro to Trees

```
class Node
{
    public:
    Node()
    {
        left = NULL;
        right = NULL;
    }

    char data;
    Node* left;
    Node* right;
};
```

```
class BinaryTree
{
    public:
    BinaryTree();
    ~BinaryTree();

    void Insert( char data );
    Node* GetNode( char data );

    private:
    Node* CreateNode( char data );
    void DestroyTree();

    Node* m_root;
    int m_size;
};
```

Our tree object will store a pointer to the **first node: The root**

The root is the first thing **Inserted** into the tree.



Intro to Trees

```
void BinaryTree::Insert( char data )
{
    if ( m_root == NULL )
    {
        // First Node
        Node* m_root = new Node;
        m_root->data = data;
    }
}
```

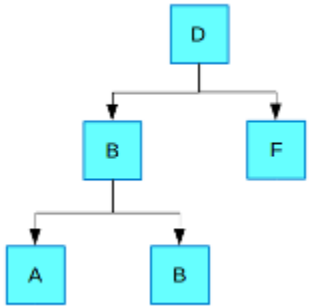
First step in an **insert** function -
If the root is NULL, then the tree is empty.

We create a **new Node**, that the root pointer will be pointing to.

We also set the value of **data** to the parameter that was passed into **Insert.**

Note #1: in the Node class' constructor, it initializes the **left** and **right** pointers to NULL.

Note #2: Since we are allocating memory with the **new** keyword, we will need to make sure to **delete** this memory in the destructor.



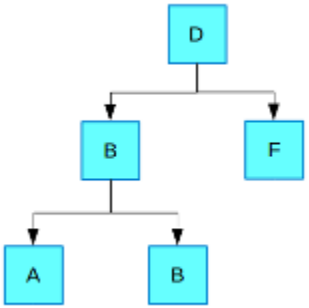
Intro to Trees

```
void BinaryTree::Insert( char data )
{
    if ( m_root == NULL )
    {
        // First Node
        Node* m_root = new Node;
        m_root->data = data;
    }
}
```

If the root isn't NULL, we need to expand this to **find a position for the new Node**, and set up the **new** Node.

Implementing this would be easiest with a recursive function.

We're going to cover this information with simple loops first, then have a lecture on recursion, then revisit searching & sorting!



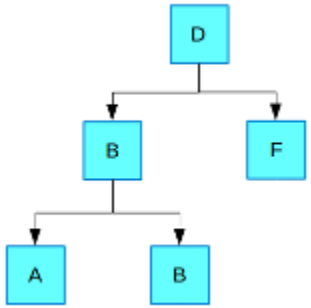
Intro to Trees

```
Node* current = m_root;
```

To begin traversing the tree, we start at the root.

```
while ( true )
{
    // LESS THAN - LEFT
    if ( data < current->data )
    {
    // GREATER THAN - RIGHT
    else if ( data > current->data )
    {
}
```

We will compare the new value to the current element's value, to figure out whether we need to go **left** or **right**.

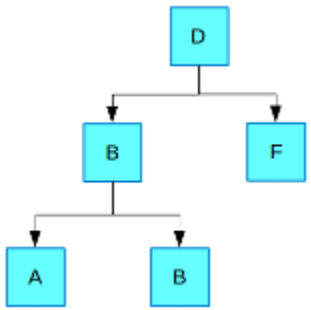


Intro to Trees

```
// LESS THAN - LEFT
if ( data < current->data )
{
    // No node here yet - found position for new node
    if ( current->left == NULL )
    {
        current->left = new Node;
        current->left->data = data;
        break;
    }
    // Already a node here - go to this node
    else
    {
        current = current->left;
    }
}
// GREATER THAN - RIGHT
else if ( data > current->data )
{
```

If we're going left, but no node is there, we've found the position for our new item.

If we're going left, and there is already a node there, we update our traversal pointer (**current**), and go back to the beginning of the loop.

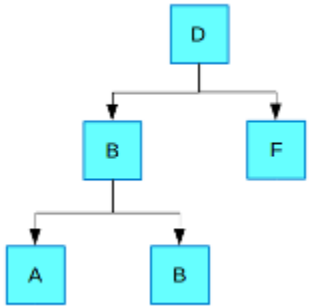


Intro to Trees

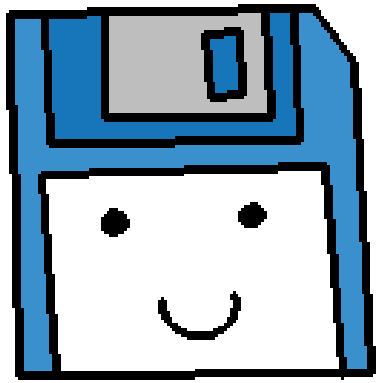
```
void BinaryTree::Insert( char data )
{
    if ( m_root == NULL )
    {
        Node* m_root = new Node;
        m_root->data = data;
    }
    else
    {
        Node* current = m_root;
        while ( true )
        {
            if ( data < current->data )
            {
                if ( current->left == NULL )
                {
                    current->left = new Node;
                    current->left->data = data;
                    break;
                }
                else
                    current = current->left;
            }
            else if ( data > current->data )
            {
                if ( current->right == NULL )
                {
                    current->right = new Node;
                    current->right->data = data;
                    break;
                }
                else
                    current = current->right;
            }
        }
    }

    m_size++;
}
```

This function would be much smaller and simpler with recursion, but hopefully this version gives you an idea of how inserting data into a tree (and keeping it ordered!) works.

Searching iteratively would work about the same, except instead of searching for a NULL space, you're comparing the node's value to the search term parameter.



Sorting Data



Sorting

Finding data in a data structure is more efficient if we have **sorted** our data ahead of time.

If the data is unsorted, we could do a linear search or some form of divide-and-conquer, but we can't do more intelligent searches based on the ordering.

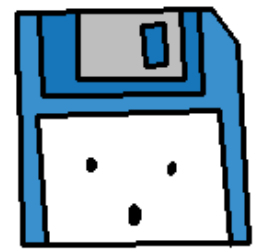
Sorting is another area where there are multiple algorithms for sorting our data, and they each have their own trade-offs.

Sorting

The Binary Tree is a data structure that is built to stay sorted, which is enforced by how items are inserted into it.

But we might also need to sort something more linear, like an array, after-the-fact – so that our search will be more efficient.

Let's try one sorting algorithm for now!



Sorting

Some sorting techniques are...

Insertion Sort

Shellsort

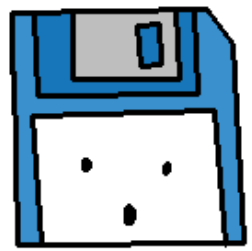
Quicksort

Bubble Sort

Heapsort

Bucketsort

Mergesort

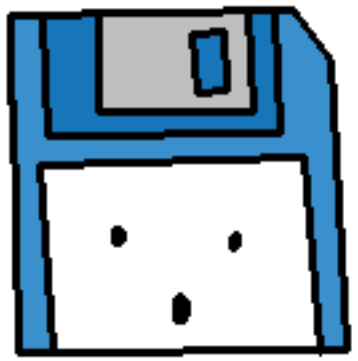


Insertion Sort

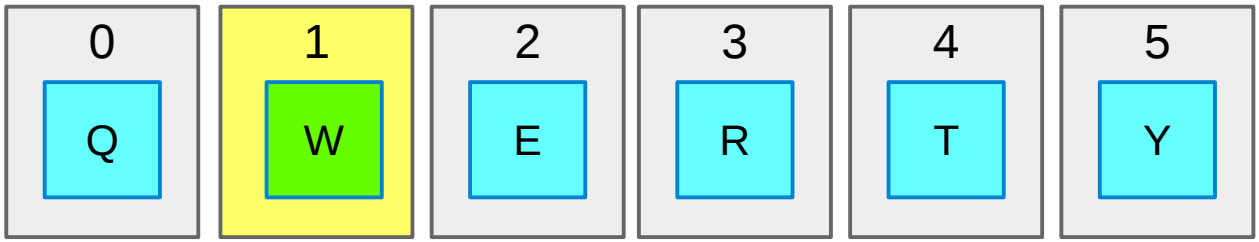
```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

With insertion sort,
it iterates over every element
of an array (except #0),
“picks it up”, and looks at
each element behind it to see
if there is a better place for it.

Let's step through the algorithm...



Insertion Sort



We start with an array of 6 characters

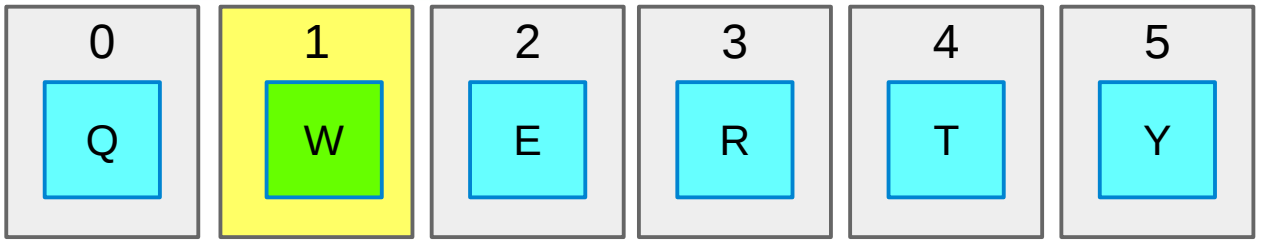
In the algorithm, we iterate through every element in order, except for #0.

p = 1

Store a backup of arr[p], which is 'W'.

```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

Insertion Sort




```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

We start with an array of 6 characters

In the algorithm, we iterate through every element in order, except for #0.

p = 1

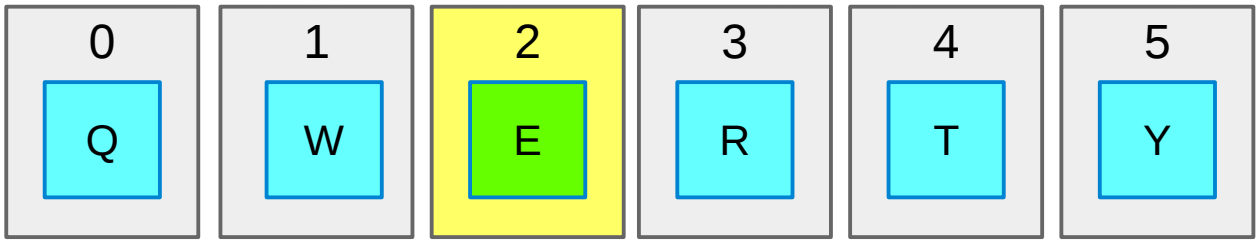
j = 1

The J loop will start at P and go until either:
J is equal to 0, or
The value at P is greater than or equal to the value at J-1.

J-1 = 0,
Arr[J-1] = Q,
Arr[P] = W,

W > Q, so we skip the loop.

Insertion Sort



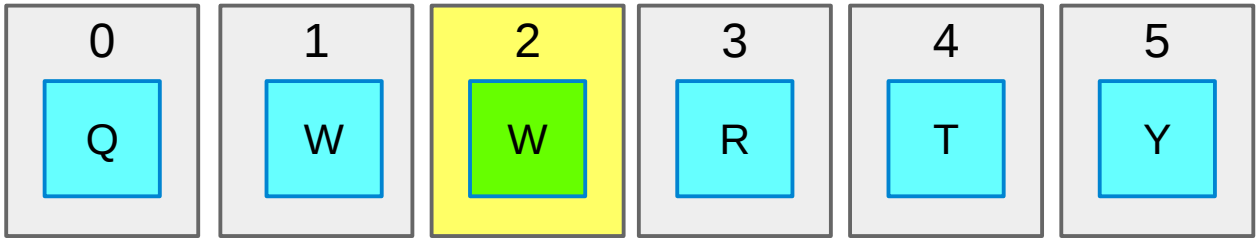
p = 2

Next, p is 2, and its value is E.

j = 2

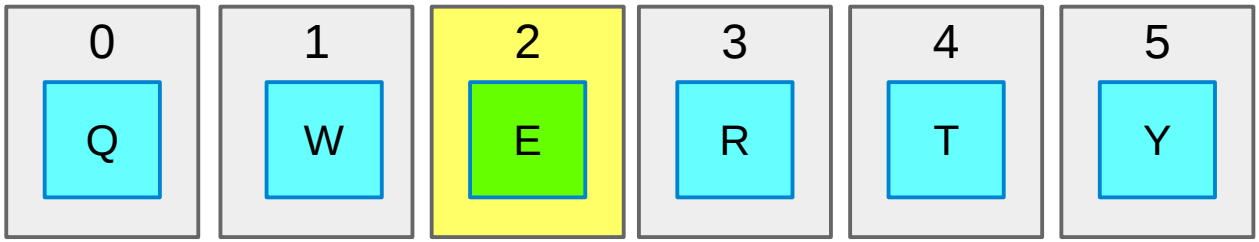
j starts at 2.

Arr[2] = arr[1] (W)




```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

Insertion Sort



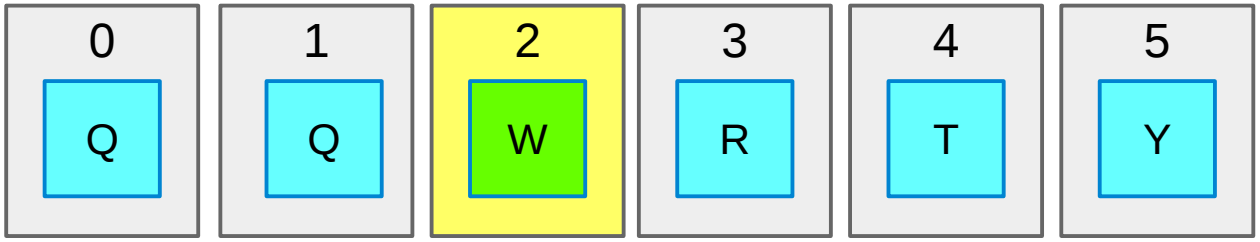
p = 2

Next, p is 2, and its value is E.

j = 1

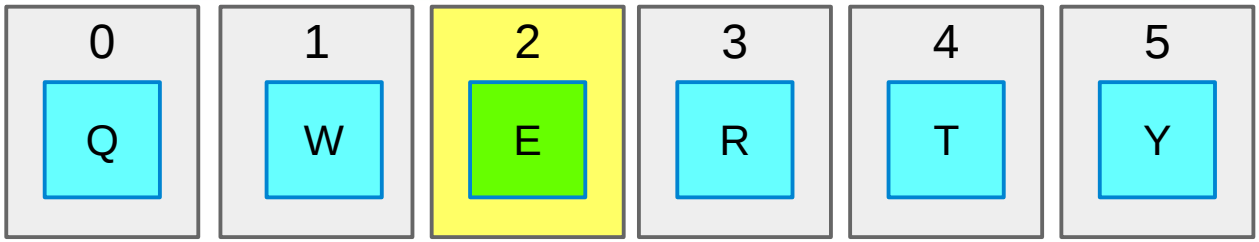
j is now 1.

Arr[1] = arr[0] (Q)




```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

Insertion Sort



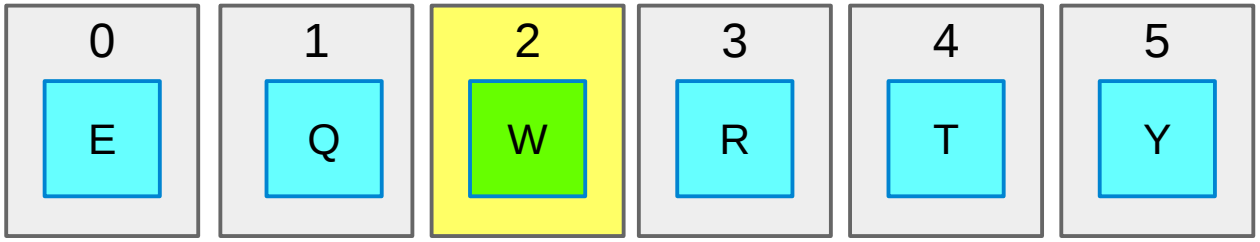
p = 2

Next, p is 2, and its value is E.

j = 0

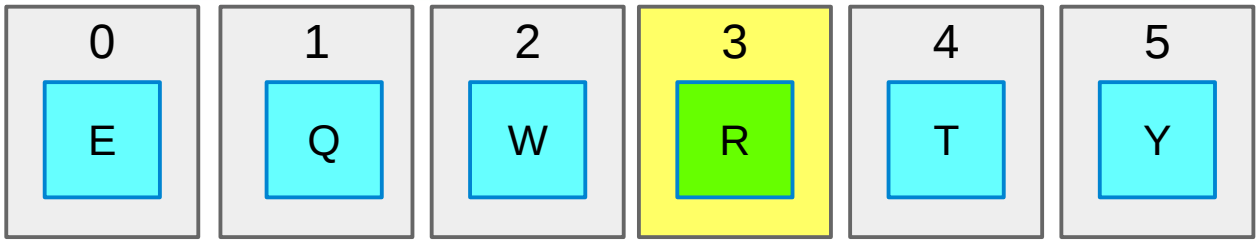
Then after the J for-loop:

Arr[0] = E




```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

Insertion Sort



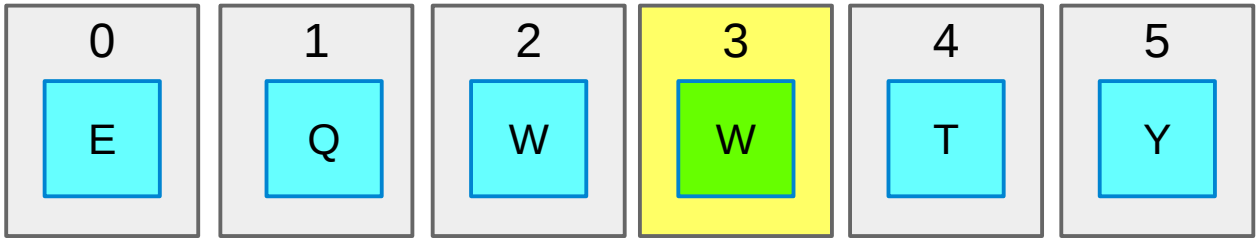
p = 3

Next, p is 3, and its value is R.

j = 3

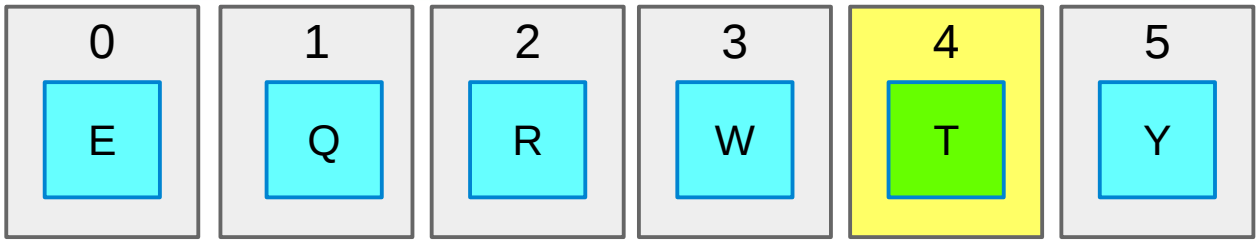
j starts at 3.

Arr[3] = arr[2] (W)




```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

Insertion Sort



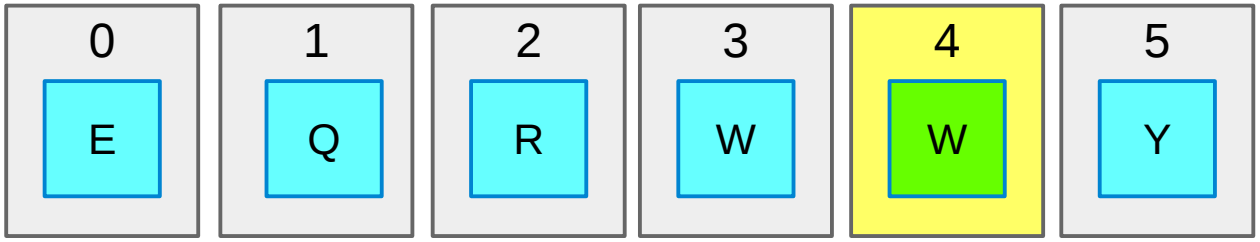
p = 4

Next, p is 4, and its value is T.

j = 3

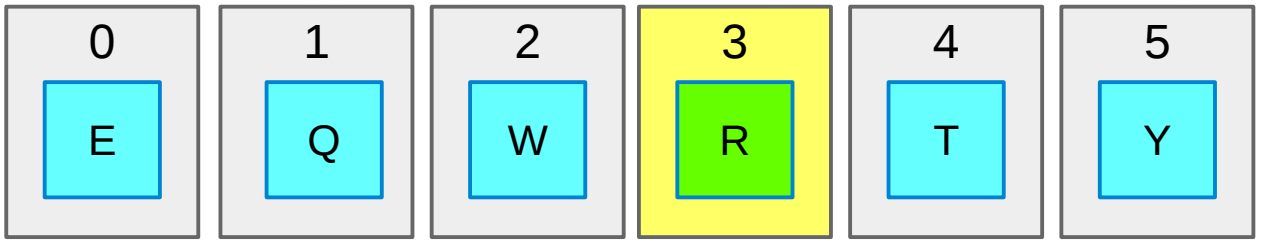
j is now 3.

Arr[2] = R, Value = T, T > R, loop stops




```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

Insertion Sort




```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

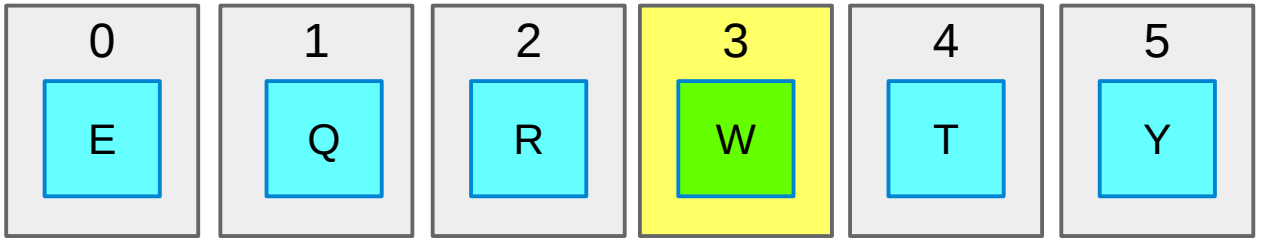
p = 3

Next, p is 3, and its value is R.

j = 2

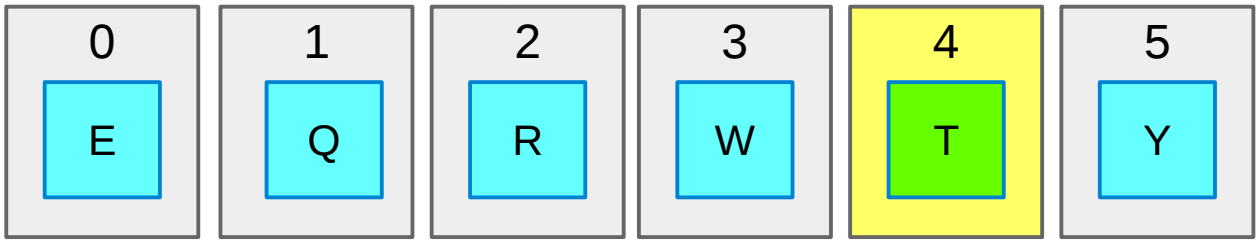
j is now 2.

Arr[J-1] = Q, Value = R, R is greater so leave loop



Arr[2] = R

Insertion Sort



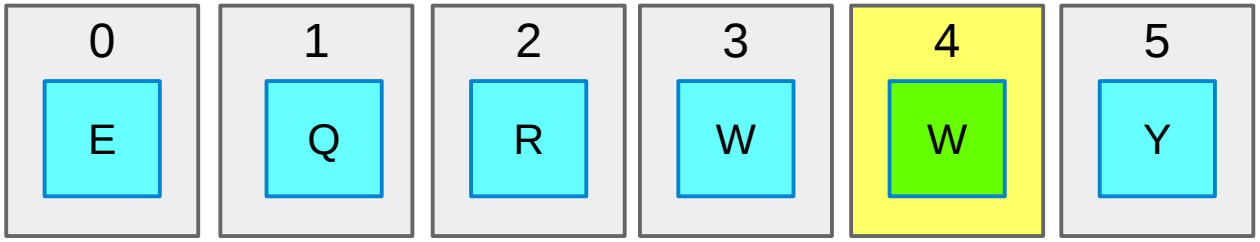
p = 4

Next, p is 4, and its value is T.

j = 4

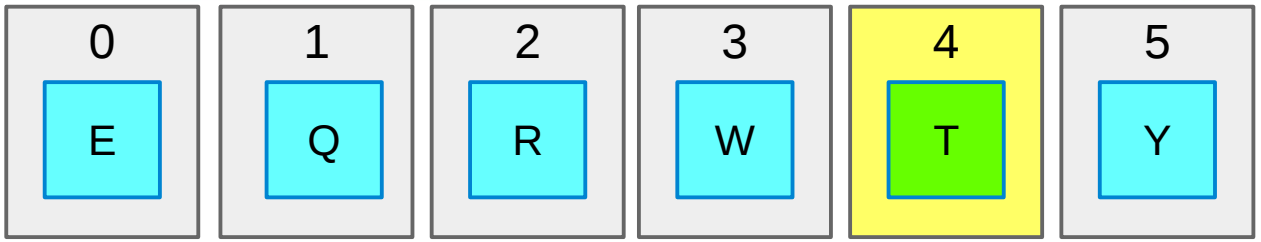
j starts at 4.

Arr[4] = arr[3] (W)




```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

Insertion Sort



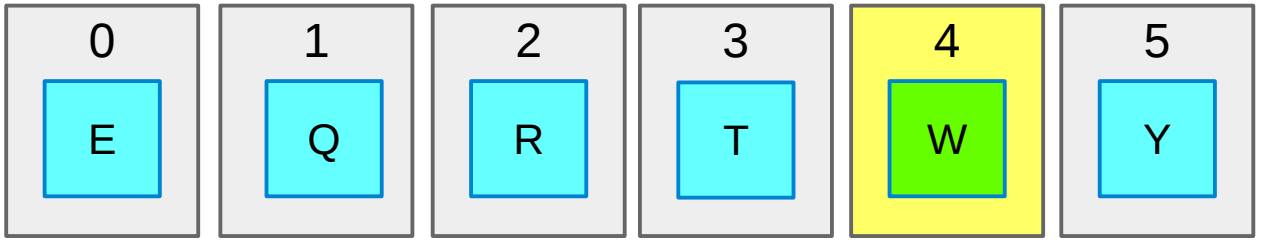
p = 4

Next, p is 4, and its value is T.

j = 3

j is now 3.

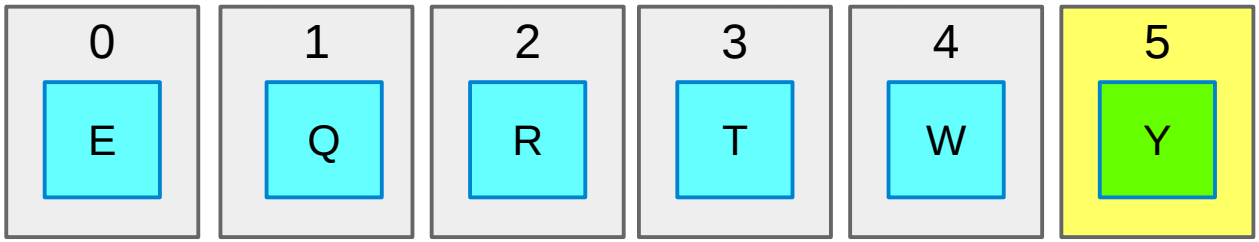
Arr[2] = R, Value = T, T > R, loop stops



Arr[3] = T

```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

Insertion Sort



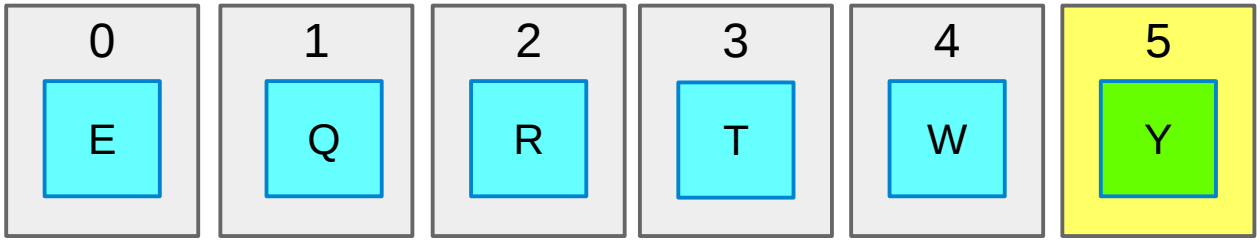
p = 5

Next, p is 5, and its value is Y.

j = 5

j starts at 5

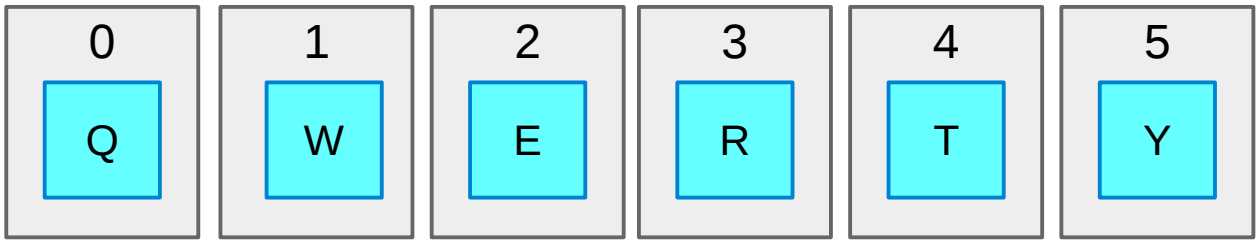
Arr[4] = T, Arr[5] = Y, Y > T, skip the loop



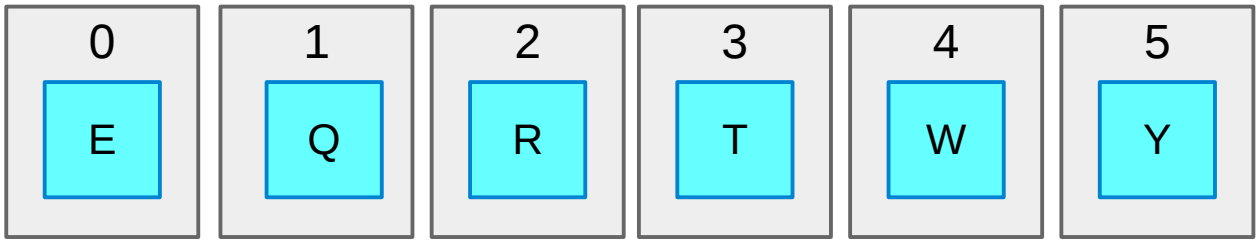

```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

Insertion Sort

Before



After



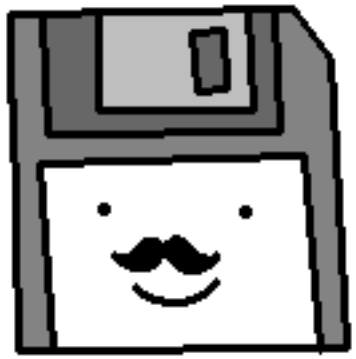

```
void InsertionSort( char arr[], int size )
{
    int j;
    for ( int p = 1; p < size; p++ )
    {
        char value = arr[p];

        for ( j = p; j > 0 && value < arr[j-1]; j-- )
        {
            arr[j] = arr[j-1];
        }
        arr[j] = value;
    }
}
```

Searching & Sorting

That was a brief introduction to searching and sorting.

We will cover more on this topic after we look at recursion!

Though for many cases, you will end up utilizing
someone *else's* sort function
(such as from the standard library)
since they will be more optimized,
and why reinvent the wheel?



If you think this is interesting,
good for you!

I'm going to go take a nap now.