Data Structures and Abstractions

Queues

Lecture 22

Queues

- Queues are ADS that emulate, for example, a queue at the movies: you can only get on the back of the queue, and off at the front of the queue.
- There are only two operations shown for a queue: [1]
  - **`Enqueue (something on to it)`**
  - **`Dequeue (something off it)`**
- Plus two query methods:
  - **`Empty ()`**
  - **`Full ()`**
- Since the last thing on is the last thing off, they are known as FIFO (First In, First Out) data structures, or sometimes LILO (Last In, Last Out).

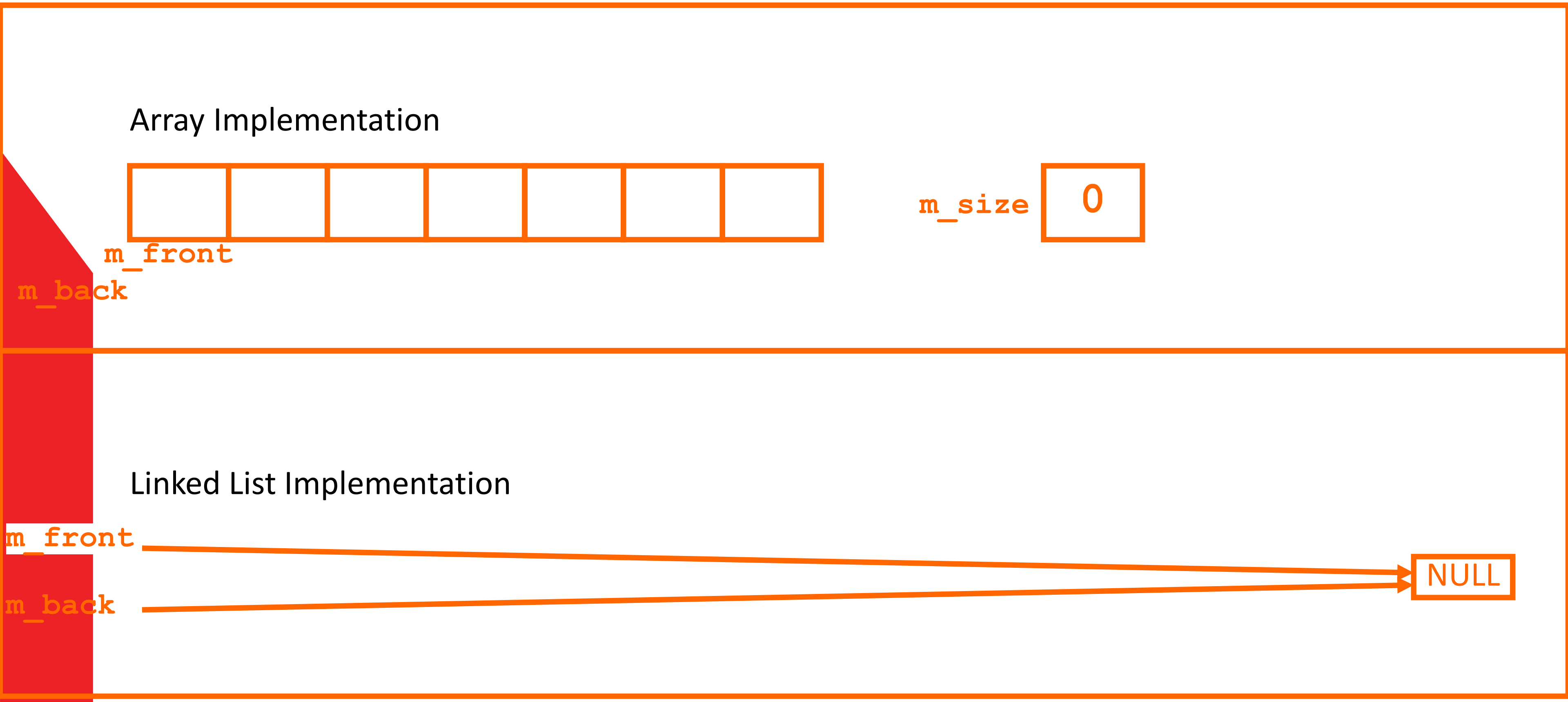
Queue Implementation

- Queues can be implemented any way you want, the encapsulation of the container used ensures that it does not matter.
- As long as it only has **`Enqueue, Dequeue, Empty`** and **`Full`**, then it is a minimal queue.
- Most commonly they are implemented using arrays, lists or an STL structure, with the STL Queue being more relevant.
- However since none of these exactly fit the required minimal abstraction we are after, they should always be encapsulated.

Error Conditions for Queues

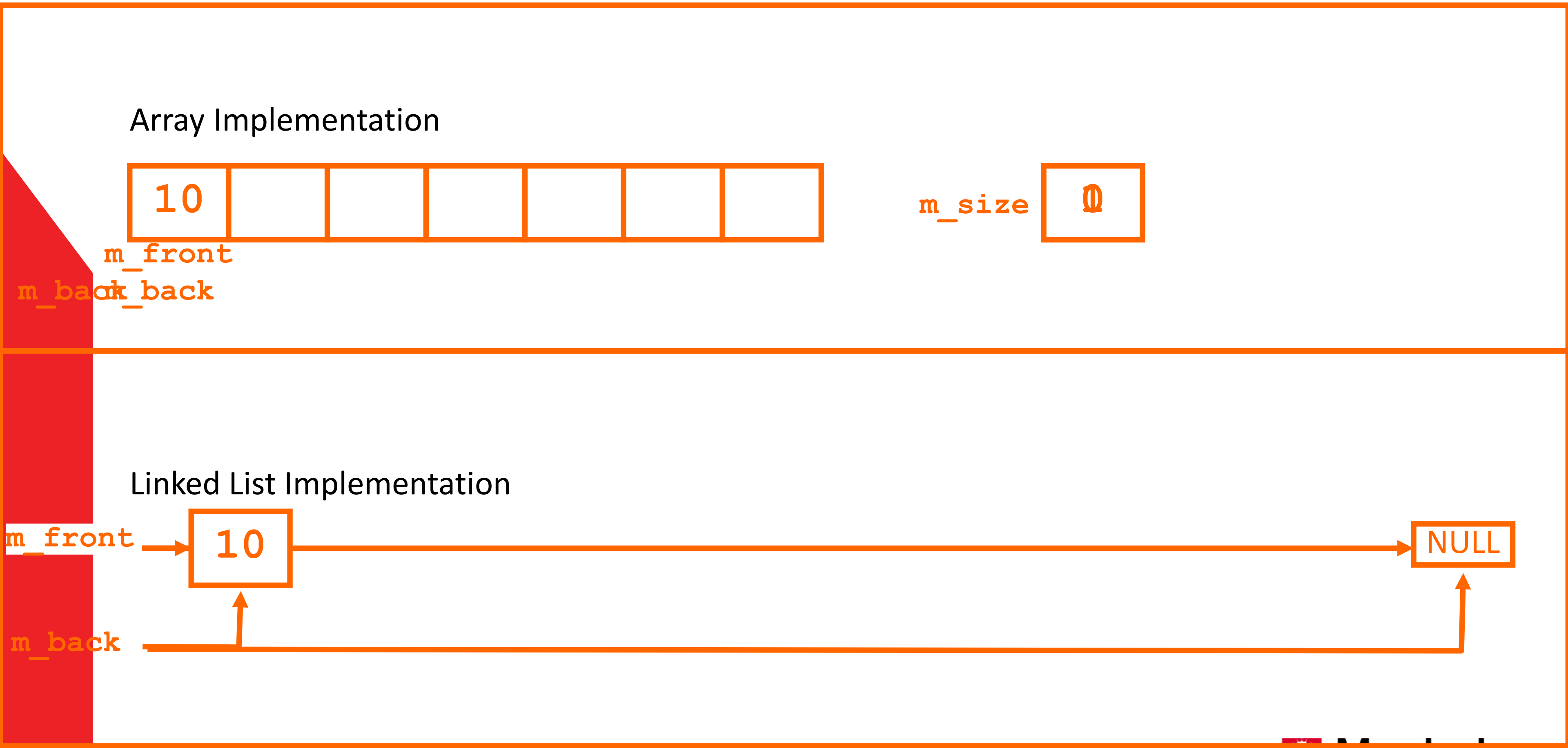
- If you try to **`Enqueue()`** onto a queue that has no free memory, then you get *overflow*.
- If you try to **`Dequeue()`** from an empty queue then you have *underflow*.
- So **`Enqueue()`** and **`Dequeue()`** return a boolean to indicate if one of these errors has occurred.
- In the animation that follows, two approaches are used.
  - The internal container is an array
  - The internal container is a linked list

Queue Example (Animation)



Queue Example (Animation)

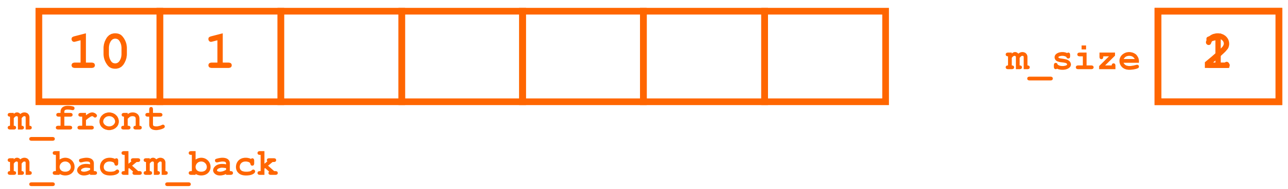
`Enqueue(10)`



Queue Example (Animation)

`Enqueue(1)`

Array Implementation

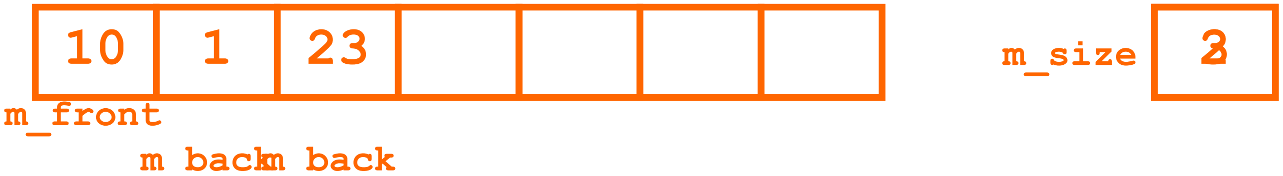


Linked List Implementation

Queue Example (Animation)

`Enqueue(23)`

Array Implementation

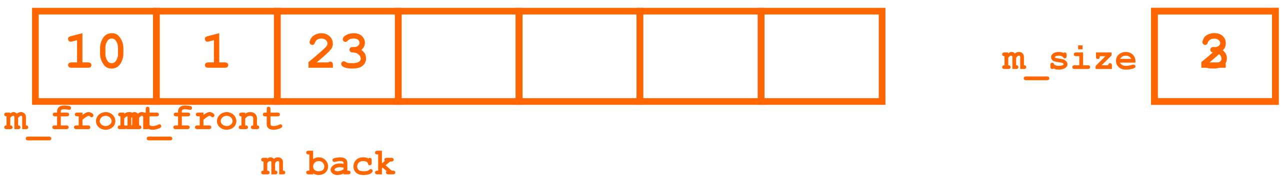


Linked List Implementation

Queue Example (Animation)

`Dequeue(num)` **`num`** 10

Array Implementation



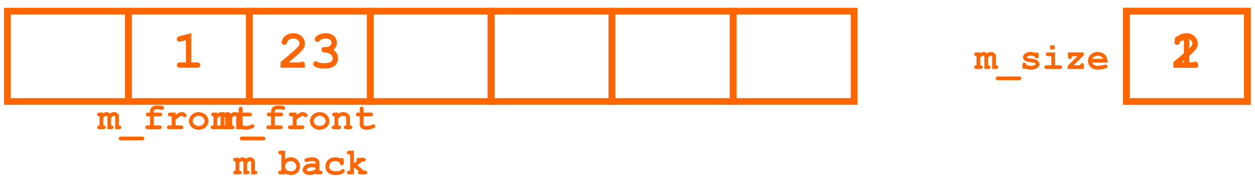
Linked List Implementation

Queue Example (Animation)

Dequeue(num)

num 1

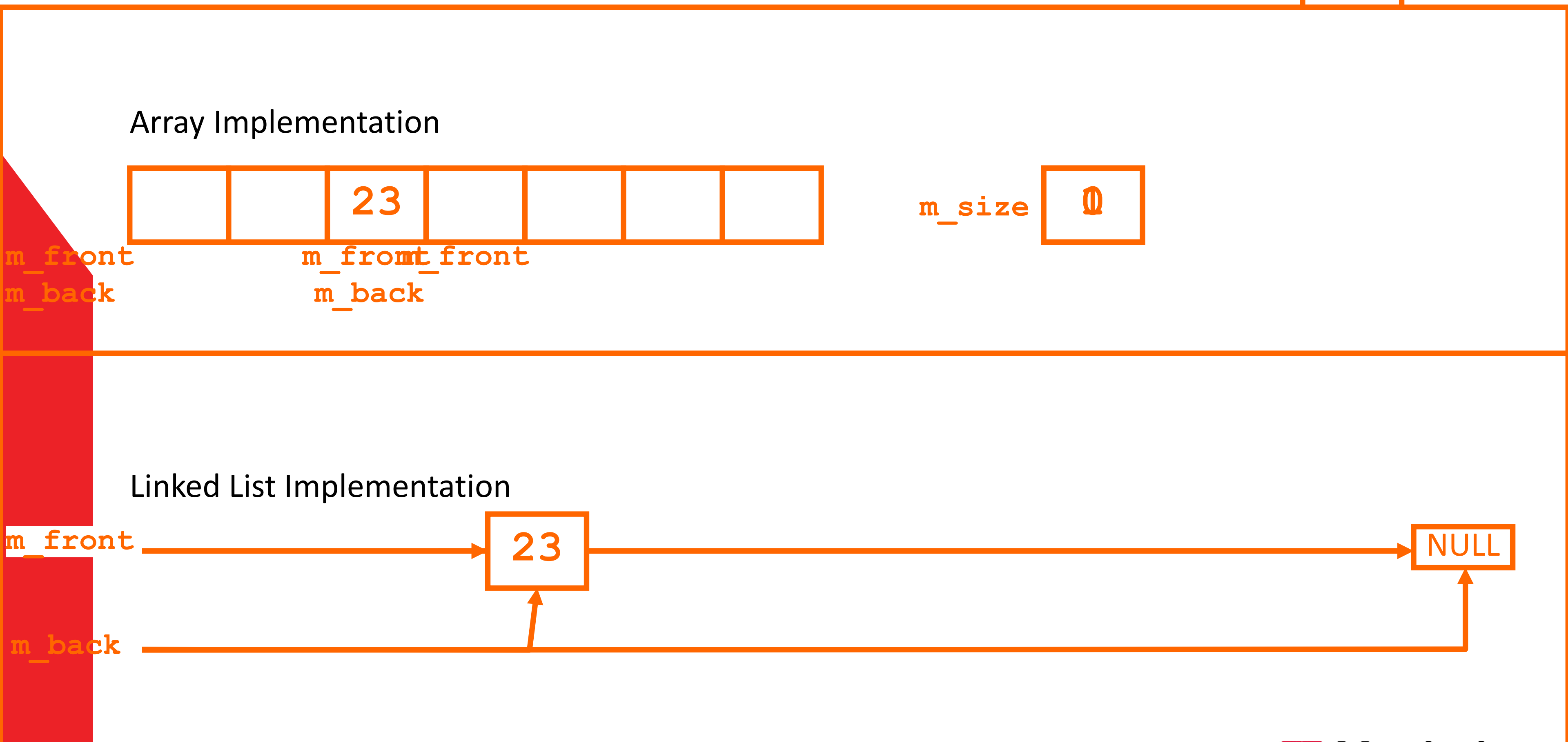
Array Implementation



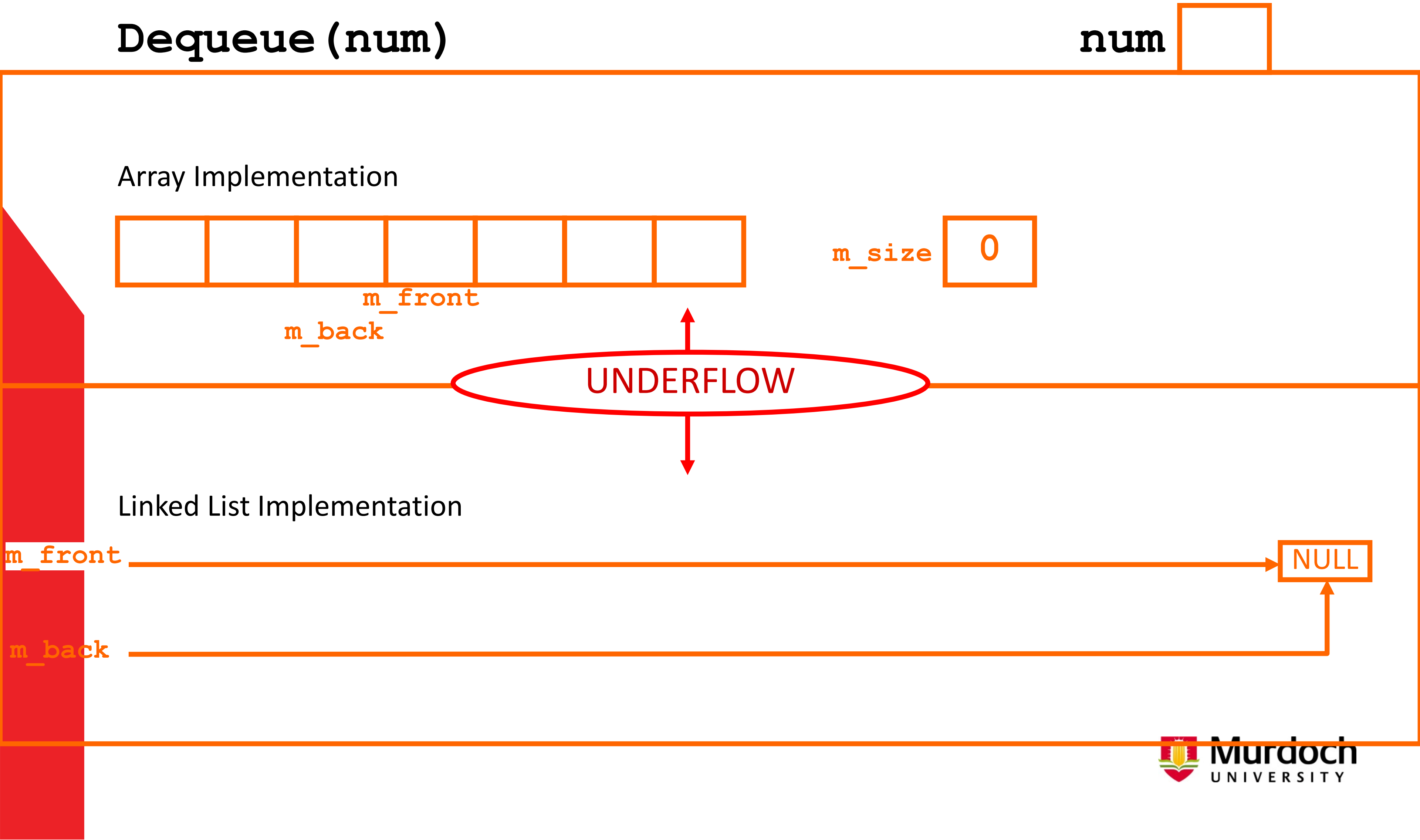
Linked List Implementation

Queue Example (Animation)

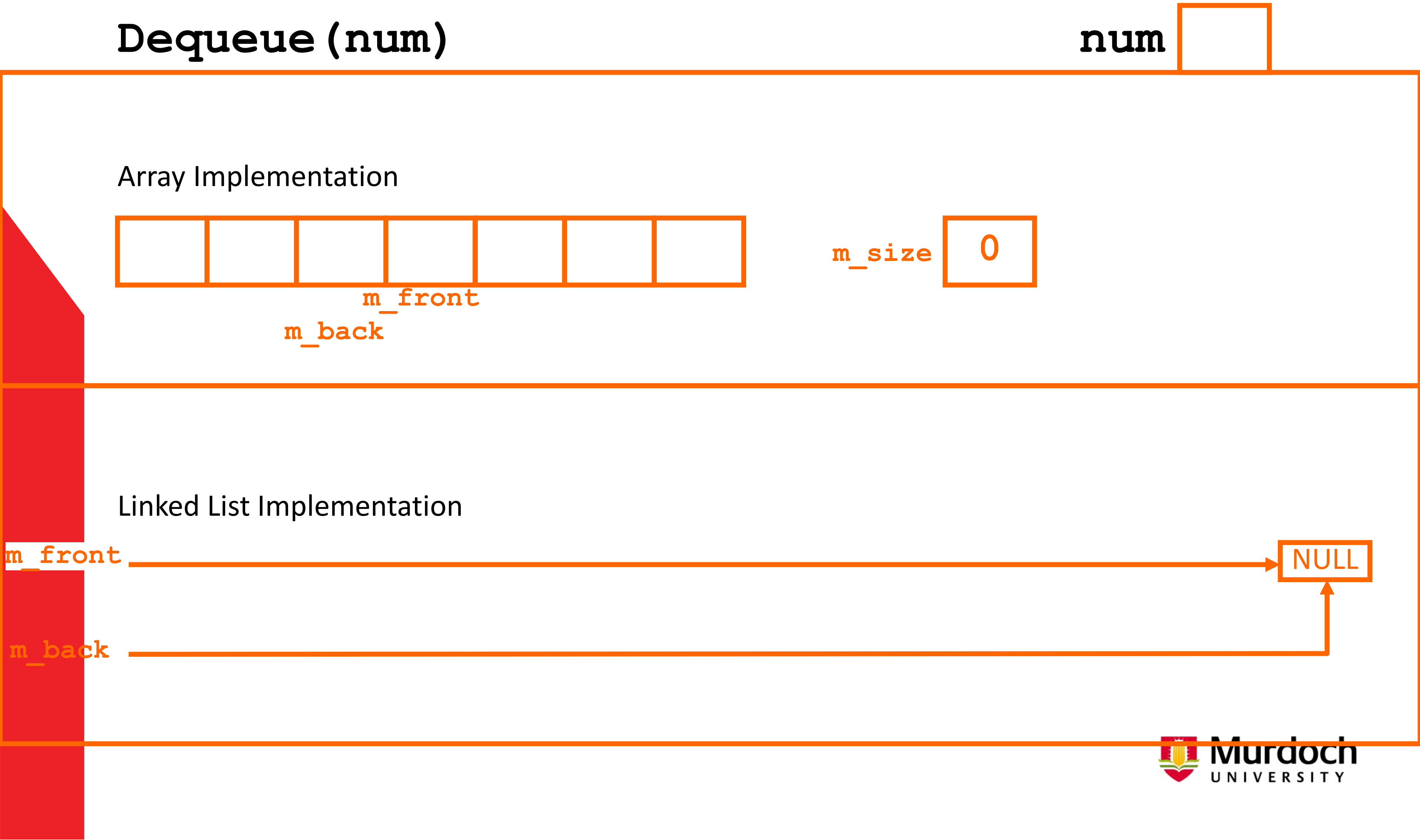
Dequeue(num) num 23



Queue Example (Animation)



Queue Example (Animation)



Queue Example (Animation)

`Enqueue(12)`

Array Implementation

12						

`m_size` 0

`m_front` `m_front`
`m_back` `m_back`

Linked List Implementation

`m_front` → 12 → NULL

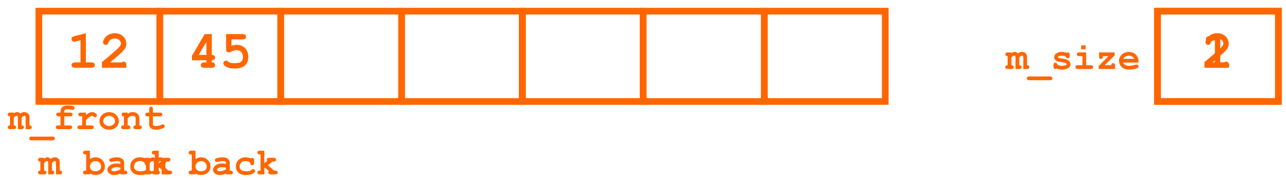
`m_back`

Murdoch UNIVERSITY

Queue Example (Animation)

Enqueue(45)

Array Implementation

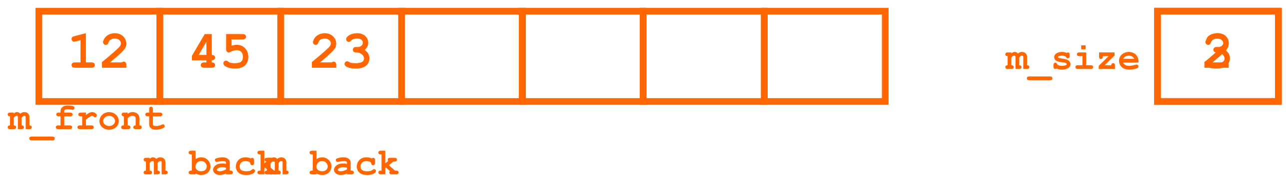


Linked List Implementation

Queue Example (Animation)

Enqueue(23)

Array Implementation

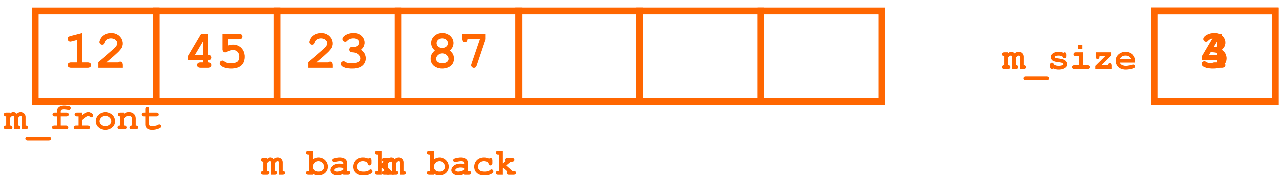


Linked List Implementation

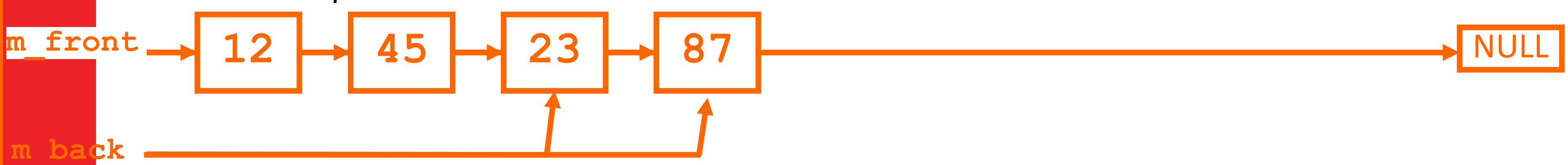
Queue Example (Animation)

Enqueue(87)

Array Implementation



Linked List Implementation



Queue Example (Animation)

`Enqueue(9)`

Array Implementation

12	45	23	87	9		

m_front

m_back m_back

m_size 5

Linked List Implementation

m_front → 12 → 45 → 23 → 87 → 9 → NULL

m_back

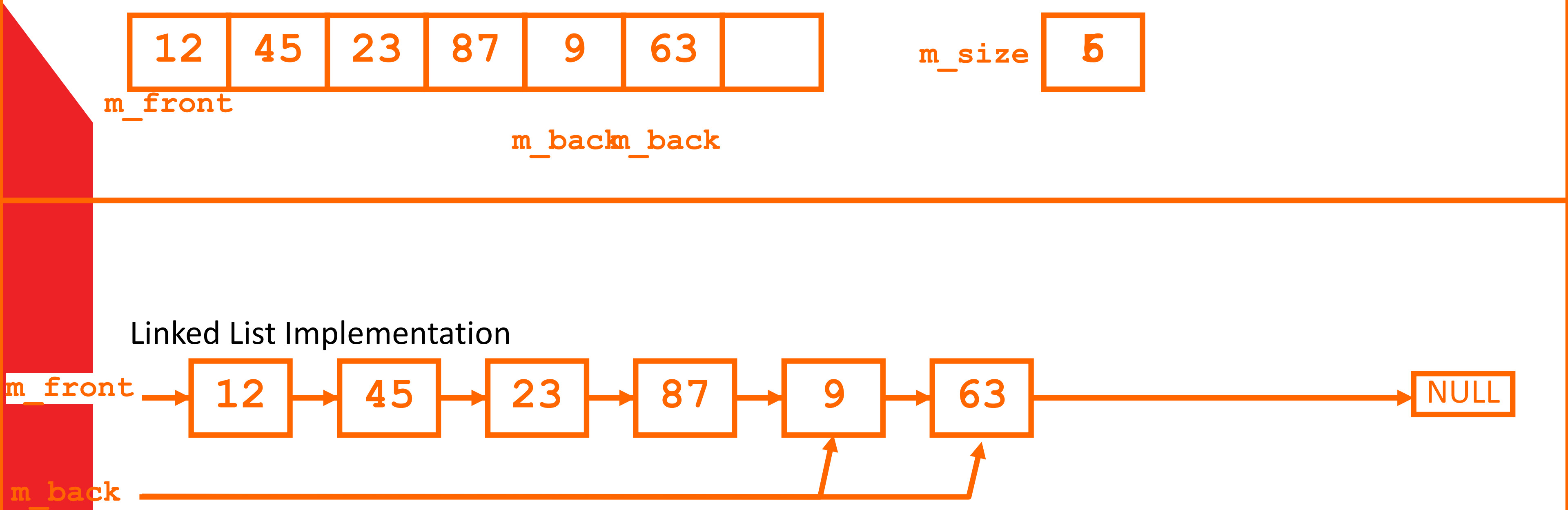
Murdoch UNIVERSITY

Queue Example (Animation)

`Enqueue(63)`

Array Implementation

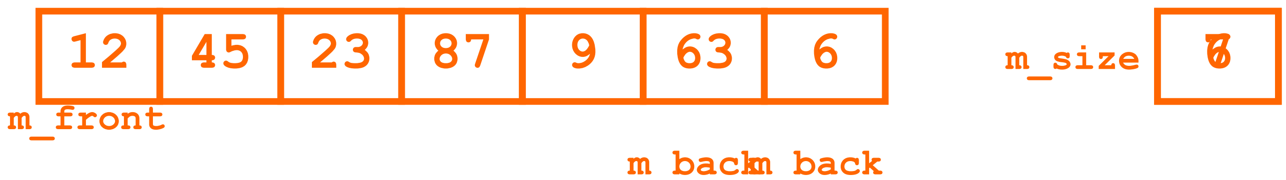
Linked List Implementation



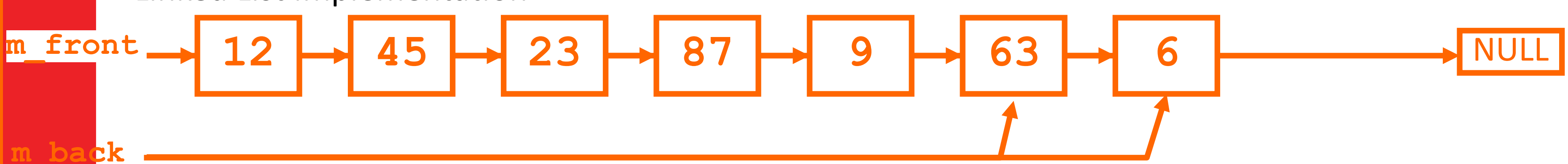
Queue Example (Animation)

`Enqueue(6)`

Array Implementation

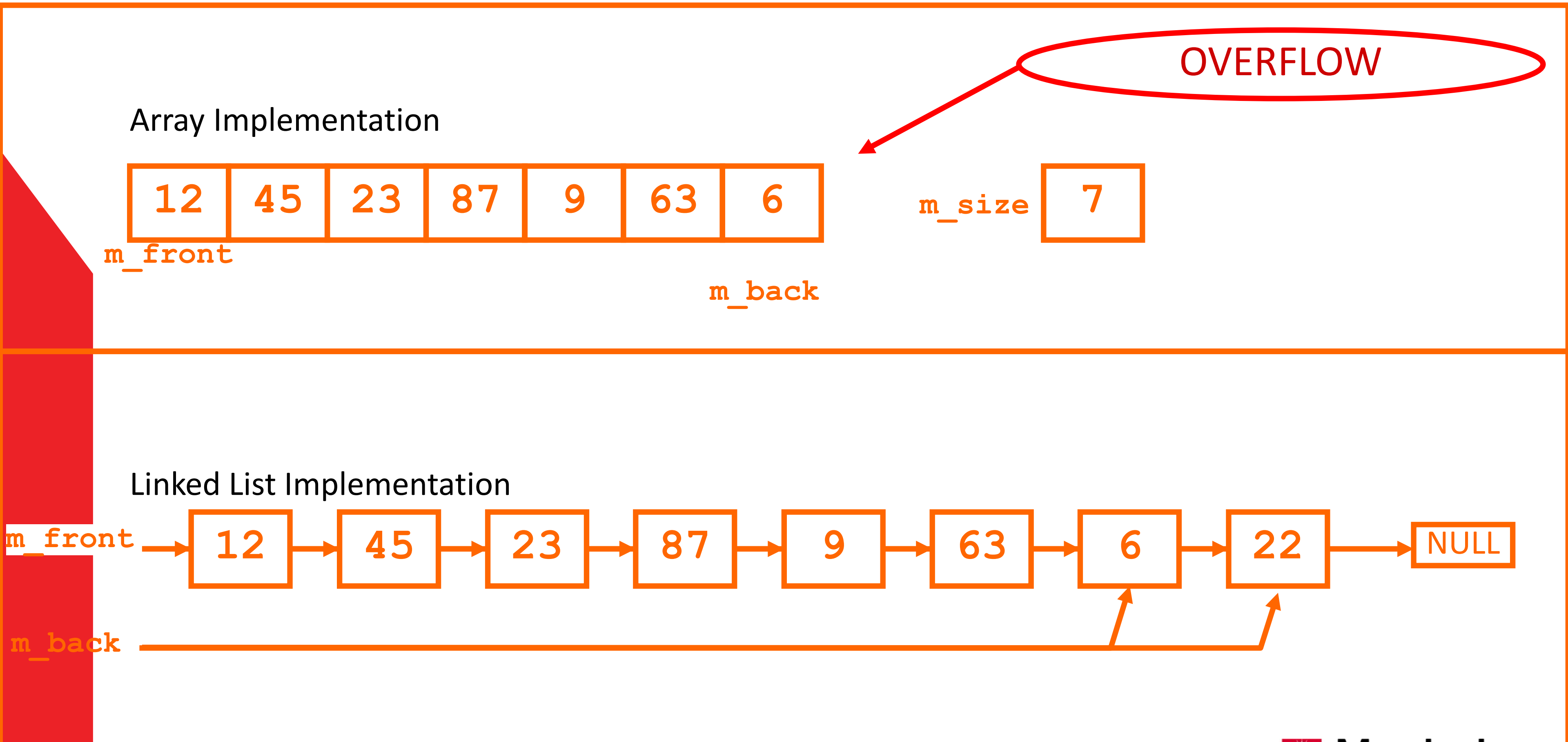


Linked List Implementation



Queue Example (Animation)

`Enqueue(22)`



Queue Example (Animation)

`Dequeue(num)`

Array Implementation

12	45	23	87	9	63	6

`m_front` `m_front`

`m_back`

`m_size` 7 6

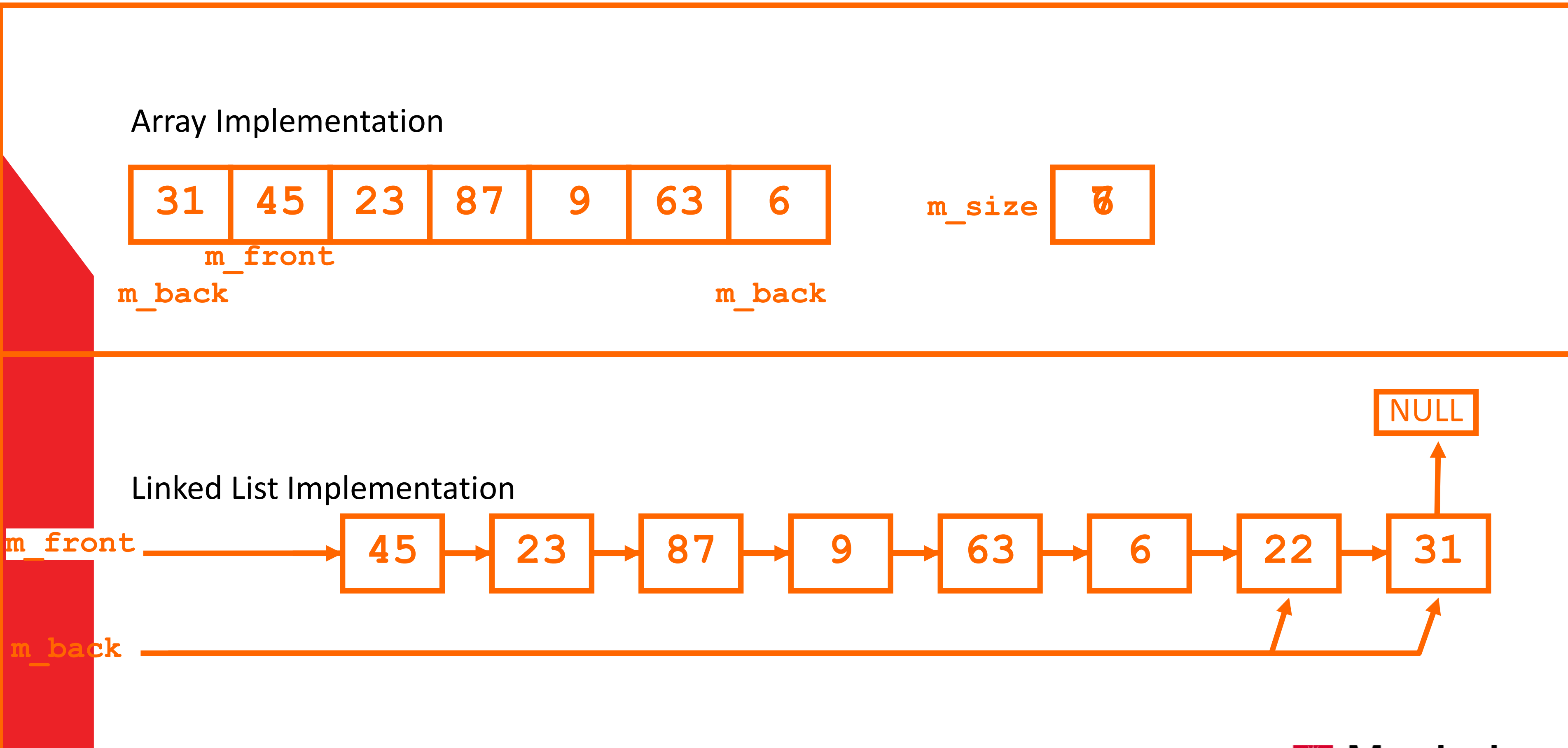
Linked List Implementation

`m_front` → 12 → 45 → 23 → 87 → 9 → 63 → 6 → 22 → NULL

`m_back` → 22

Queue Example (Animation)

`Enqueue(31)`



END

Array Enqueue Algorithm

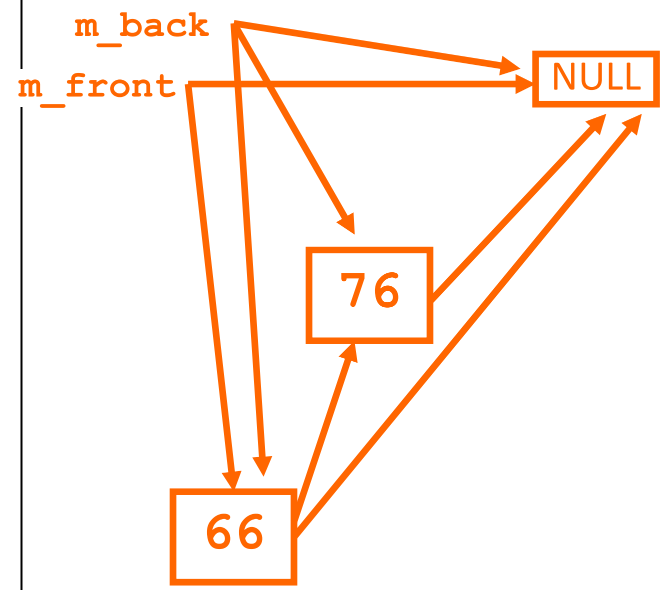
- ENQUEUE (DataType data): boolean
- IF m_size >= ARRAY_SIZE-1
- return FALSE
- ELSE
- Increment m_size
- Increment m_back MOD ARRAY_SIZE [1]
- Place data at position m_back
- return TRUE
- ENDIF
- END Enqueue

Array Dequeue Algorithm

- DEQUEUE (DataType data): boolean
- IF m_size == 0
- return FALSE
- ELSE
- data = data at position m_front
- Increment m_front MOD ARRAY_SIZE
- Decrement m_size
- IF m_size == 0
- m_front = -1
- m_back = -1
- ENDIF
- return TRUE
- ENDIF
- END Dequeue

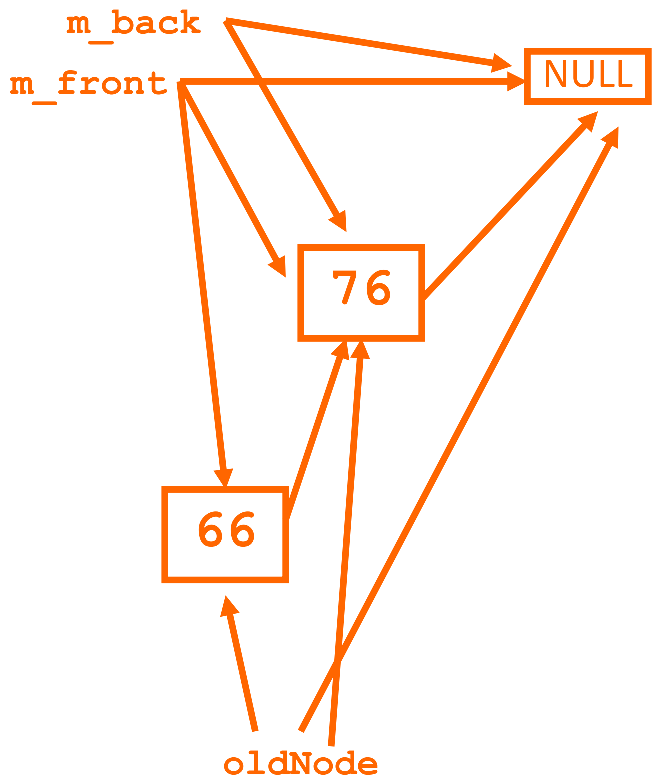
Linked List Enqueue Algorithm

- ENQUEUE (DataType data): boolean
- IF there is memory on the heap
- Get newNode from the heap
- IF m_front is NULL
- m_front = newNode
- m_back = newNode
- ELSE
- m_back.next = newNode
- m_back = newNode
- ENDIF
- return TRUE
- ELSE
- return FALSE
- ENDIF
- END Enqueue



Linked List Dequeue Algorithm

```
DEQUEUE (DataType data): boolean
    IF m_front == NULL
        return FALSE
    ELSE
        data = m_front.data
        oldNode = m_front
        IF m_back == m_front
            m_front = NULL
            m_back = NULL
        ENDIF
        m_front = oldNode.next
        release oldNode memory
        oldNode = NULL
        return TRUE
    ENDIF
END Dequeue
```



Using the STL

- The other possibility is to use one of the STL structures.
- If using a vector or list, then the algorithms above barely change.
- However, remember that the structure *must* still be encapsulated in a class, otherwise it will not have just the **`Enqueue()`** and **`Dequeue()`** that it is supposed to have.
- Finally, there is the STL queue class, (requiring <queue>), which is obviously the best STL class to use.
  - However, even this should be encapsulated, because it does not conform to the standard queue description! [1]

Murdoch UNIVERSITY

Non Standard Features of the STL queue

1. Its Enqueue method is called **`push()`** [1]
2. Its Dequeue method is called **`pop()`**.
3. Its **`pop()`** method, only removes the data, it does *not* pass it back to the calling method.
4. In fact there is a **`front()`** method which returns a *reference* to the front of the queue.
5. Neither **`dequeue()`**, **`front()`** nor **`enqueue()`** return a boolean: overflow and underflow must be checked separately.

- Therefore it is best that the STL queue must be encapsulated. [2]

Murdoch UNIVERSITY

Queue Header File using STL queue

```
// Queue.h
//
// Queue class
//
// See actual code provided in the zip file
//----------------------------------------

#ifndef MY_QUEUE
#define MY_QUEUE

//----------------------------------------

#include <queue> // for the STL queue
#include <iostream>
using namespace std;

//----------------------------------------
```

```
template <class T>
class Queue // minimal and complete
{
public:
        Queue () {};
        ~Queue () {};
        bool Enqueue(const T &data);
        bool Dequeue (T &data);
        bool Empty () const {return m_queue.empty();}
private:
        queue<T> m_queue; // encapsulates STL queue
};
```

```
//-----------------------------------------
// It is a template, so we have to put all the code
//   in the header file
//-----------------------------------------

template<class DataType>
bool Queue<DataType>::Enqueue(const DataType &data)
{
        bool okay = true;
        try
        {
            m_queue.push(data); // calls STL queue method
        }
        catch (...)
        {
            okay = false;
        }

        return okay;
}
```

```cpp
//--------------------------------------------

template<class DataType>
bool Queue<DataType>::Dequeue(DataType &data)
{
    if (m_queue.size() > 0)
    {
        data = m_queue.front();
        m_queue.pop();
        return true;
    }
    else
    {
        return false;
    }
}

//--------------------------------------------

#endif
```

Simple (but interesting) Example of Queue Use

```
// IntQueueTest Program
//
// Version
//   original by - Nicola Ritter
//   modified by smr
//
//-----------------------------------------------------

#include "Queue.h"
#include <iostream>
#include <ctime>
using namespace std;

//-----------------------------------------------------

const int EVENT_COUNT = 20;
const int MAX_NUM = 100;

//-----------------------------------------------------

typedef Queue<int> IntQueue;
typedef Queue<float> FloatQueue;

//-----------------------------------------------------
```

```
•     void DoEvents ();
•     void AddNumber (IntQueue &aqueue);
•     void DeleteNumber (IntQueue &aqueue);
•     void TestOverflow();

•     //------------------------------------------------------

•     int main()
•     {
•         DoEvents ();

•         cout << endl;
•         system("Pause");
•         cout << endl;

•         TestOverflow();

•         cout << endl;
•         return 0;
•     }

•     //------------------------------------------------------
```

```
•        void DoEvents ()
•        {
•                IntQueue aqueue;

•                // Seed random number generator
•                srand (time(NULL));

•                for (int count = 0; count < EVENT_COUNT; count++)
•                {
•                    // Choose a random event
•                    int event = rand() % 5;

•                    // Do something based on that event type, biasing
•                    //   it towards Adding
•                    if (event <= 2) // event = 0, 1 or 2
•                    {
•                            AddNumber (aqueue);
•                    }
•                    else // event = 3 or 4
•                    {
•                            DeleteNumber (aqueue);
•                    }
•                }
•        // aqueue is local so destructor for aqueue is called when routine finishes.
•        }

•        //-----------------------------------------------------
```

Murdoch UNIVERSITY

```
void AddNumber (IntQueue &aqueue)
{
    // Get a random number
    int num = rand() % (MAX_NUM+1);

    // Try adding it, testing if the aqueue was full
    if (aqueue.Enqueue(num))
    {
        cout.width(3);
        cout << num << " added to the queue" << endl;
    }
    else
    {
        cout.width(3);
        cout << "Overflow: could not add " << num << endl;
    }
}

//-------------------------------------------------------
```

```
void DeleteNumber (IntQueue &aqueue)
{
        int num;
        if (aqueue.Dequeue(num))
        {
            cout.width(3);
            cout << num << " deleted from the queue" << endl;
        }
        else
        {
            cout << "IntQueue is empty, cannot delete" << endl;
        }
}

//------------------------------------------------
```

```
void TestOverflow()
{
        Queue<double> mqueue;

        int count = 0;

        // Keeping adding numbers until we run out of space, will take //time
        while (mqueue.Enqueue(count))
        {
            count++;
            cout << "Count is " << count << endl;
        }

}

//----------------------------------------------------
```

Screen Output

- IntQueue is empty, cannot delete
- 79 added to the queue
- 79 deleted from the queue
- IntQueue is empty, cannot delete
- 2 added to the queue
- 2 deleted from the queue
- IntQueue is empty, cannot delete
- 72 added to the queue
- 72 deleted from the queue
- 88 added to the queue
- 88 deleted from the queue
- 22 added to the queue
- 5 added to the queue
- 22 deleted from the queue
- 37 added to the queue
- 46 added to the queue
- 74 added to the queue
- 58 added to the queue
- 5 deleted from the queue
- 37 deleted from the queue

- Press any key to continue . . .

```
Count is 1
Count is 2
Count is 3
Count is 4
Count is 5
Count is 6
Count is 7
Count is 8
Count is 9
Count is 10
Count is 11
Count is 12
Count is 13
Count is 14
Count is 15
Count is 16
Count is 17
. . .
```

At 300,000 I stopped: it was just too boring

Advantages of Implementations

- It is assumed for each of the containers below, that the Queue encapsulates it in its own class.

Array	Linked List	list/vector/deque	STL queue
Available in all languages.	Full memory control	Easy to code	Easiest to code
	Memory 'never' runs out. [1]	Memory 'never' runs out. [1]	Memory 'never' runs out. [1]

Murdoch UNIVERSITY

Disadvantages of Implementations

- It is assumed for each of the containers below, that the Queue encapsulates it in its own class.

Array	Linked List	list/vector/deque	STL queue
Can run out of space easily.	Difficult to code as it uses pointers.	Excess code sitting ‘behind’ the implementation, increasing the size of the program.	Excess code sitting ‘behind’ the implementation, increasing the size of the program.
Messy to code, because m_back ends up in front of m_front		Available in some languages like Java, C++. [1]	Available in some languages like Java, C++. [1]

Murdoch UNIVERSITY

Readings

- **Textbook: Stacks and Queues, entire section on Queues**

- **STL Queue: http://en.cppreference.com/w/cpp/container/queue**

- **For more details of Queues with some level of language independence, see the reference book, *Introduction to Algorithms* section on "Stacks and Queues" in the chapter on "Elementary Data Structures". You will see how removed the STL queue is from the abstract queue we are after. https://prospero.murdoch.edu.au/record=b2794699~S10**

Murdoch UNIVERSITY

Stack Example Animation

Lecture 23

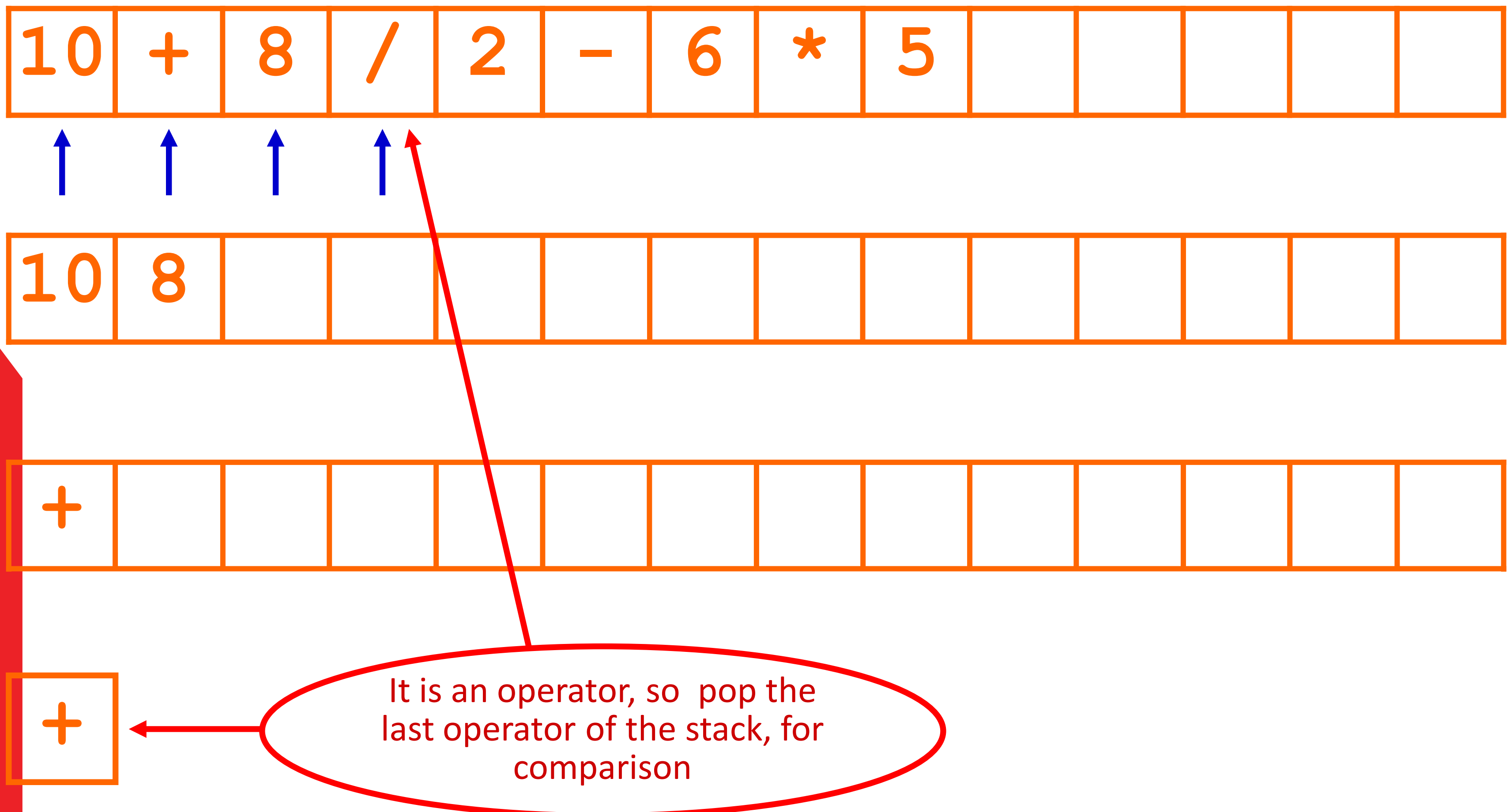
Stack Calculator Animation

- Convert the animation that follows into an algorithm
- Implement the algorithm
- Design the solution carefully

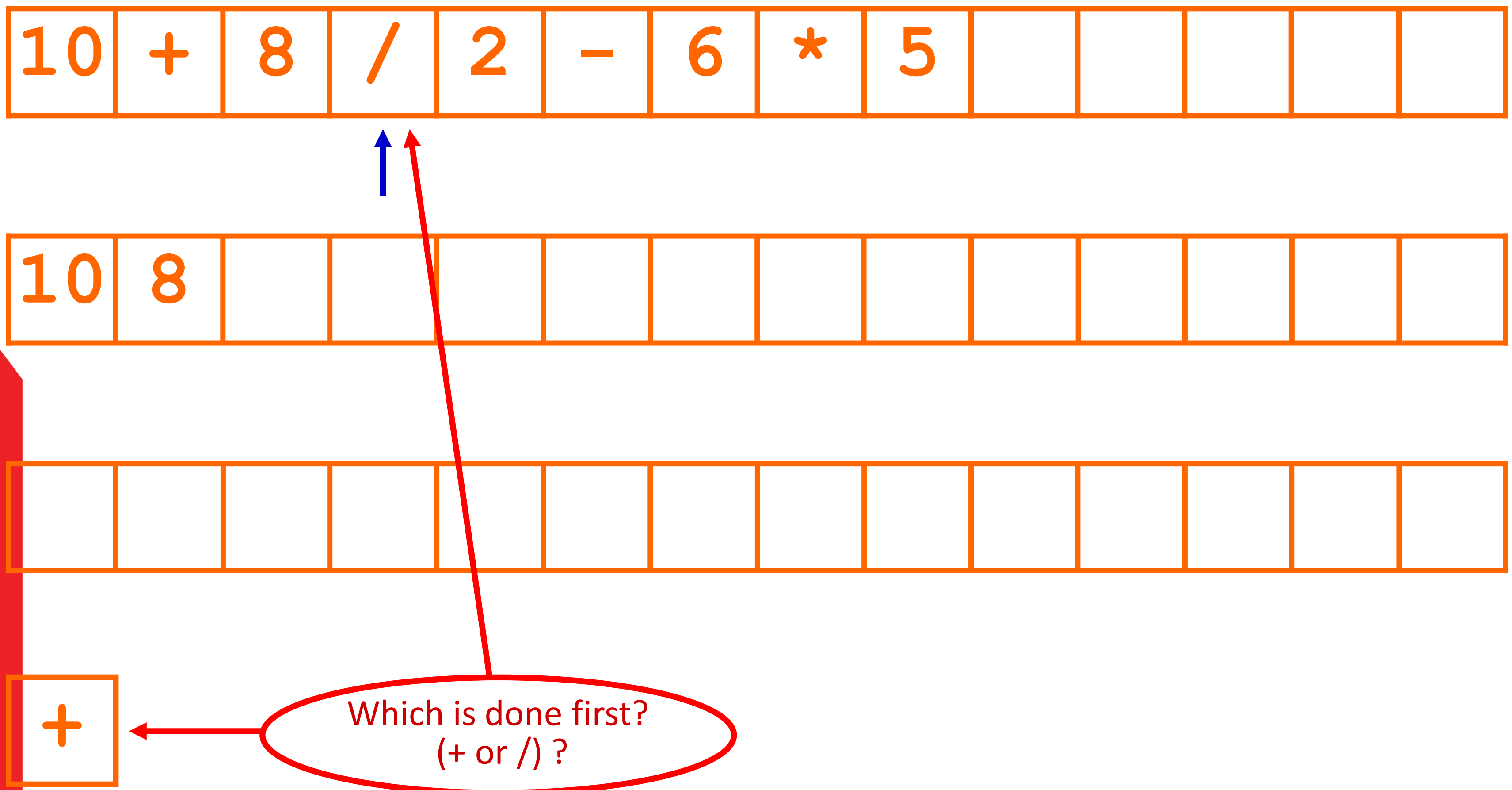
Stack Calculator Animation

10	+	8	/	2	-	6	*	5					

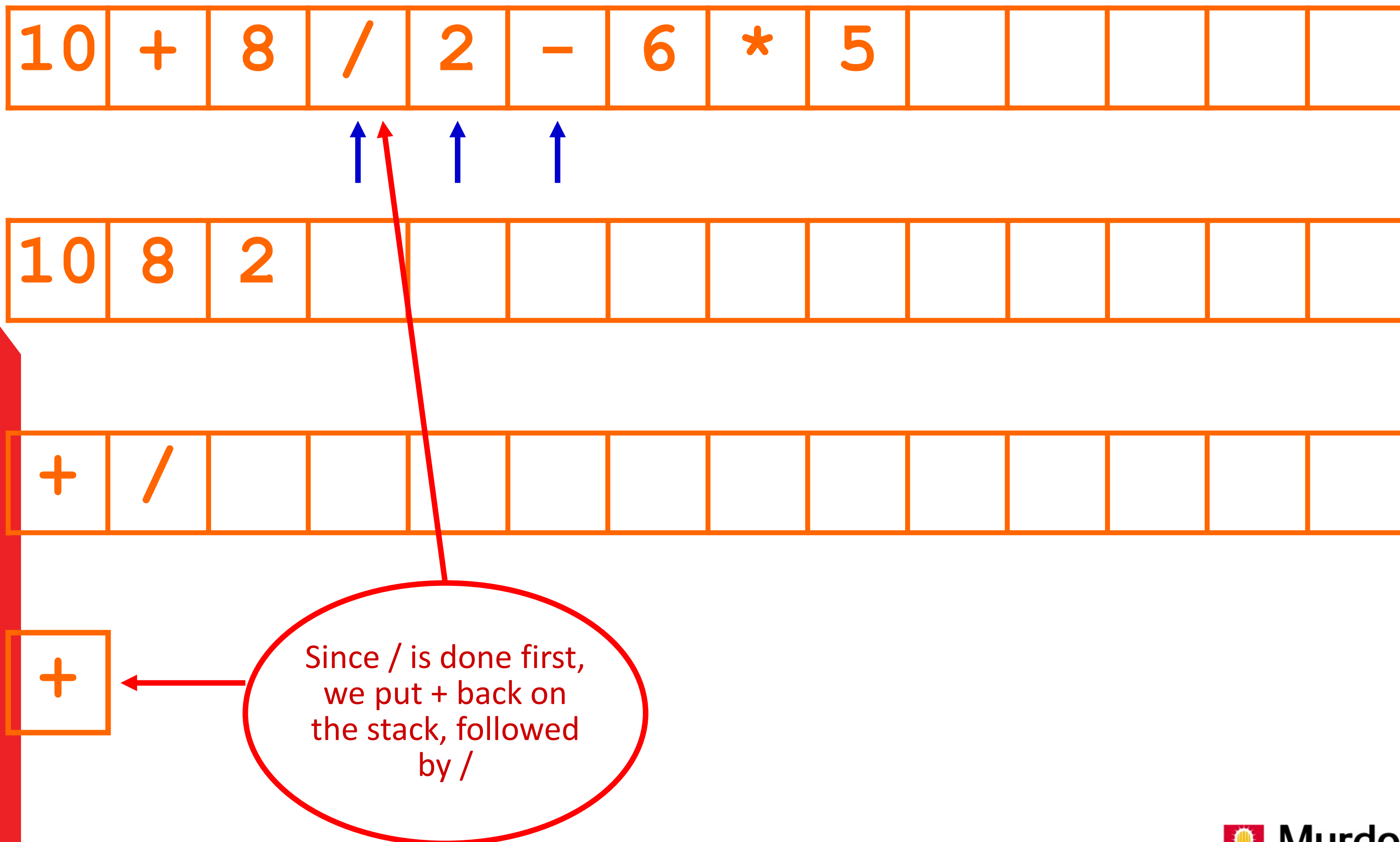
Stack Calculator Animation



Stack Calculator Animation



Stack Calculator Animation



Stack Calculator Animation

10	+	8	/	2	-	6	*	5					

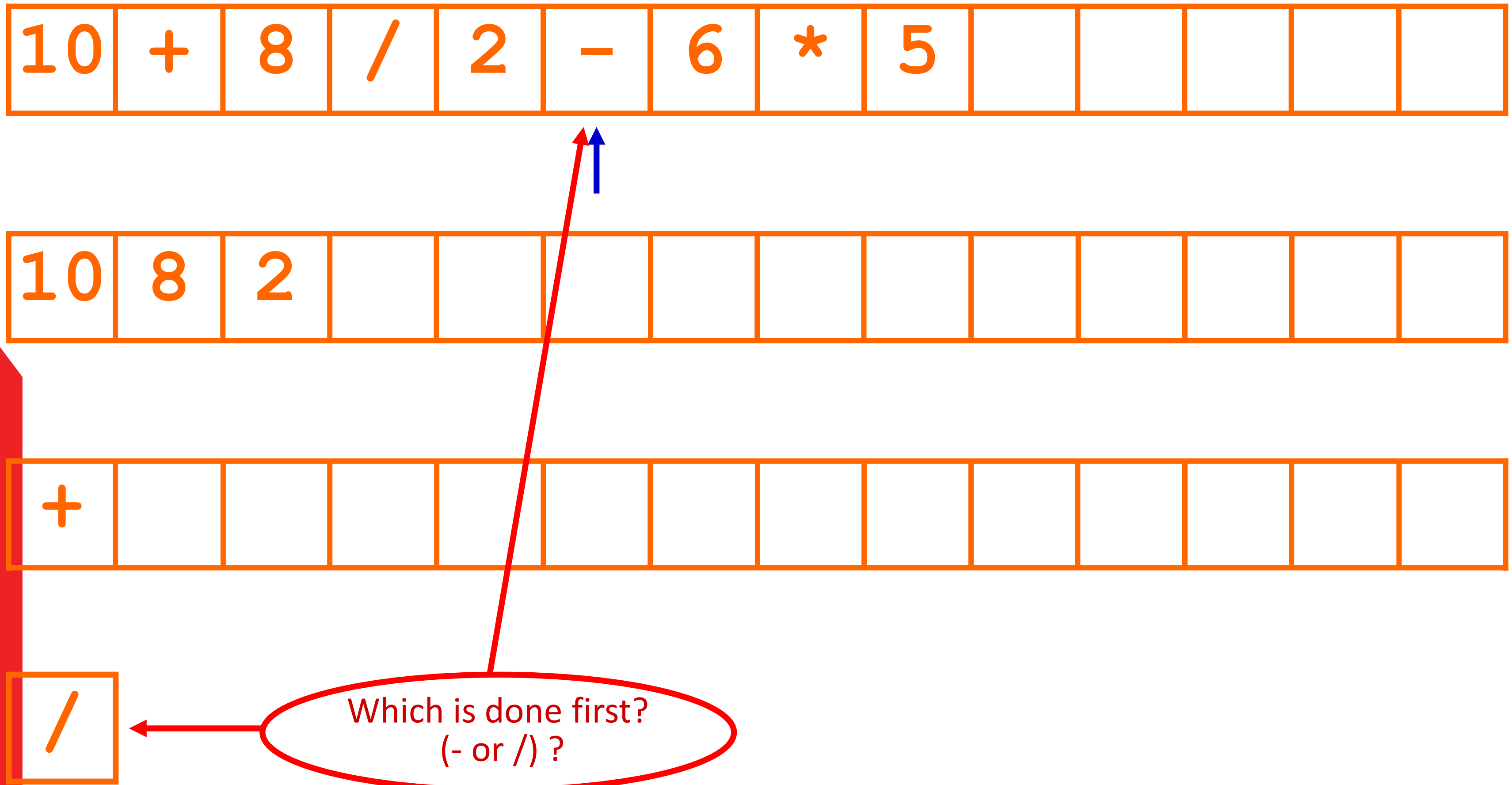
10	8	2											

+	/												

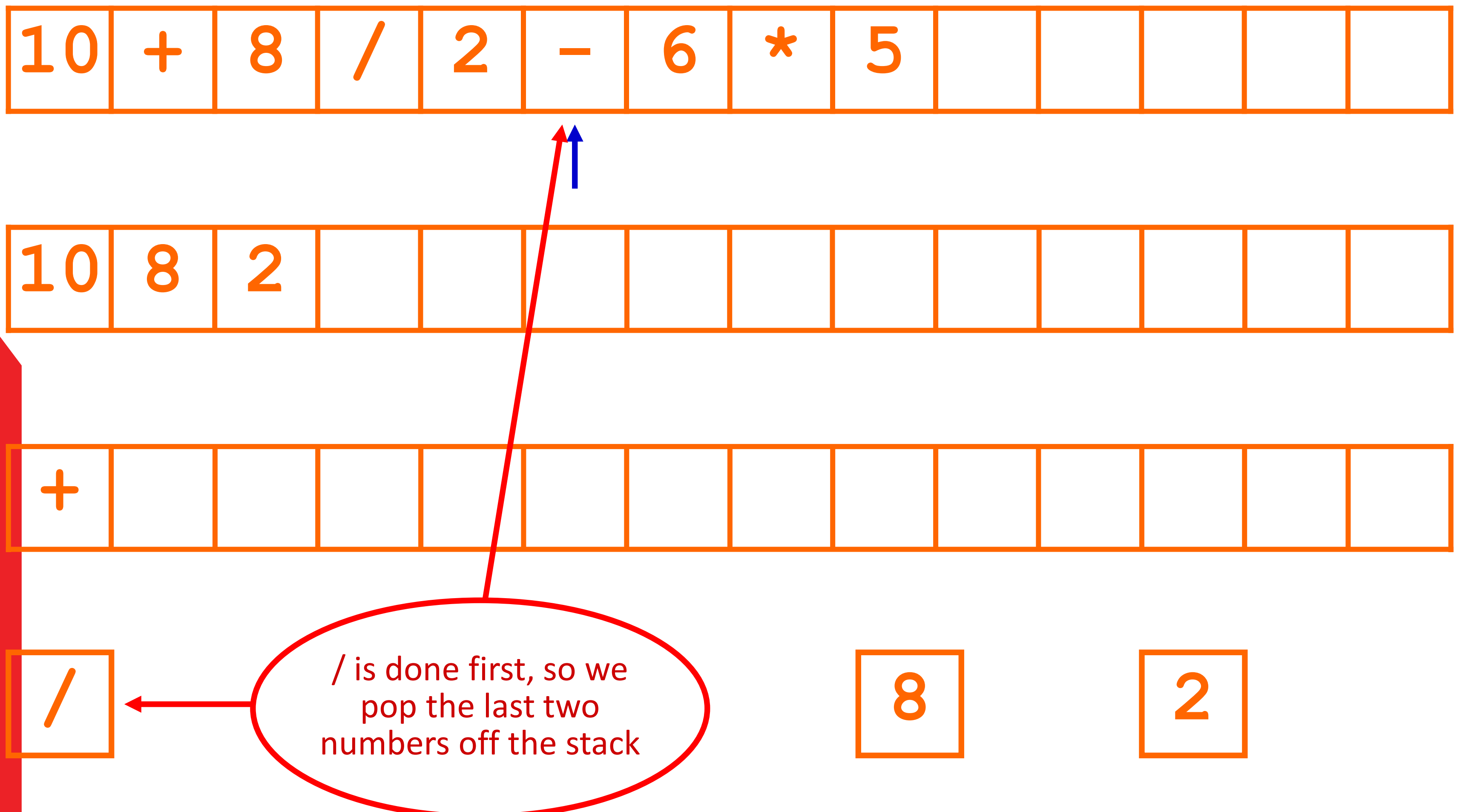
/

It is an operator, so pop the last operator of the stack, for comparison

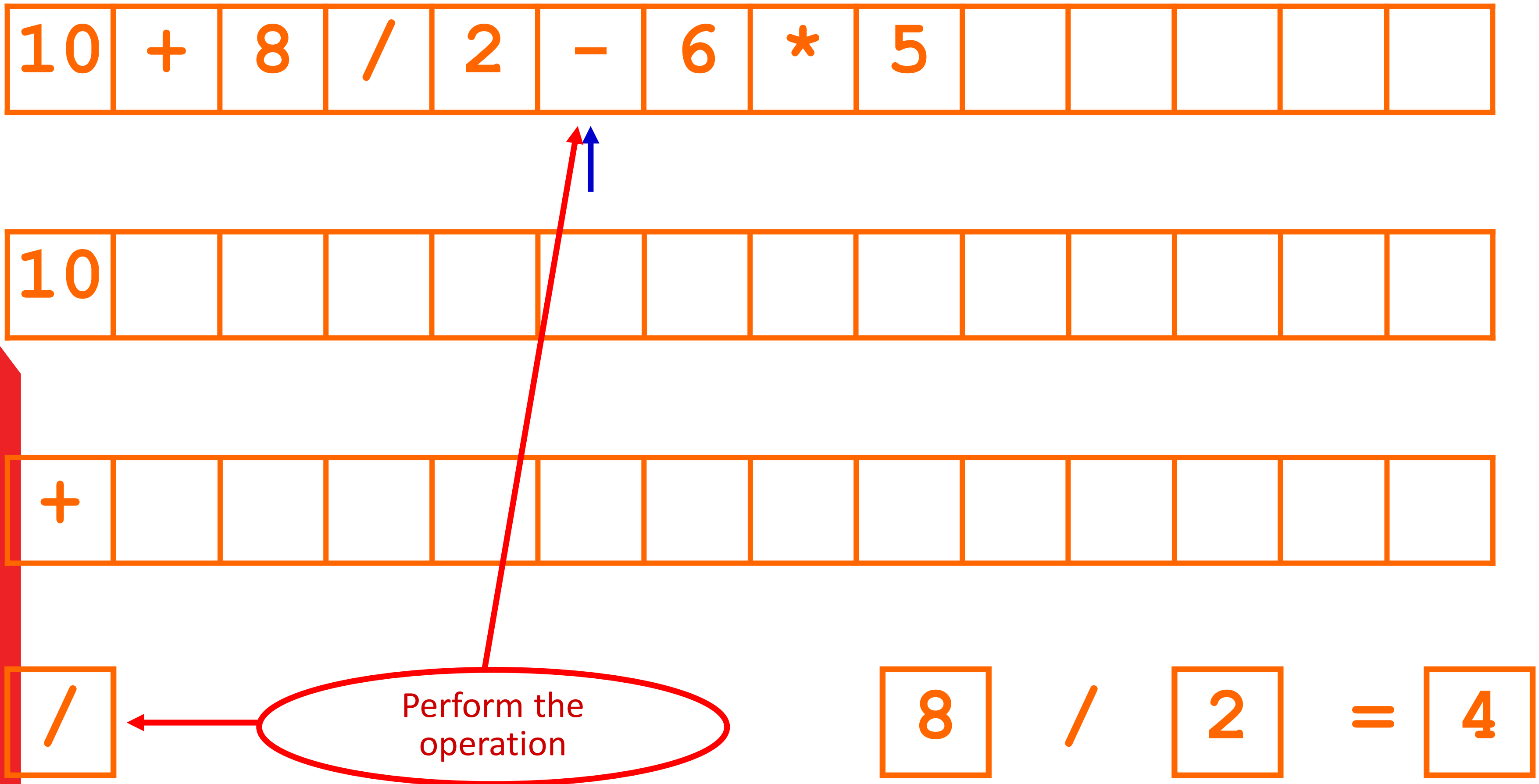
Stack Calculator Animation



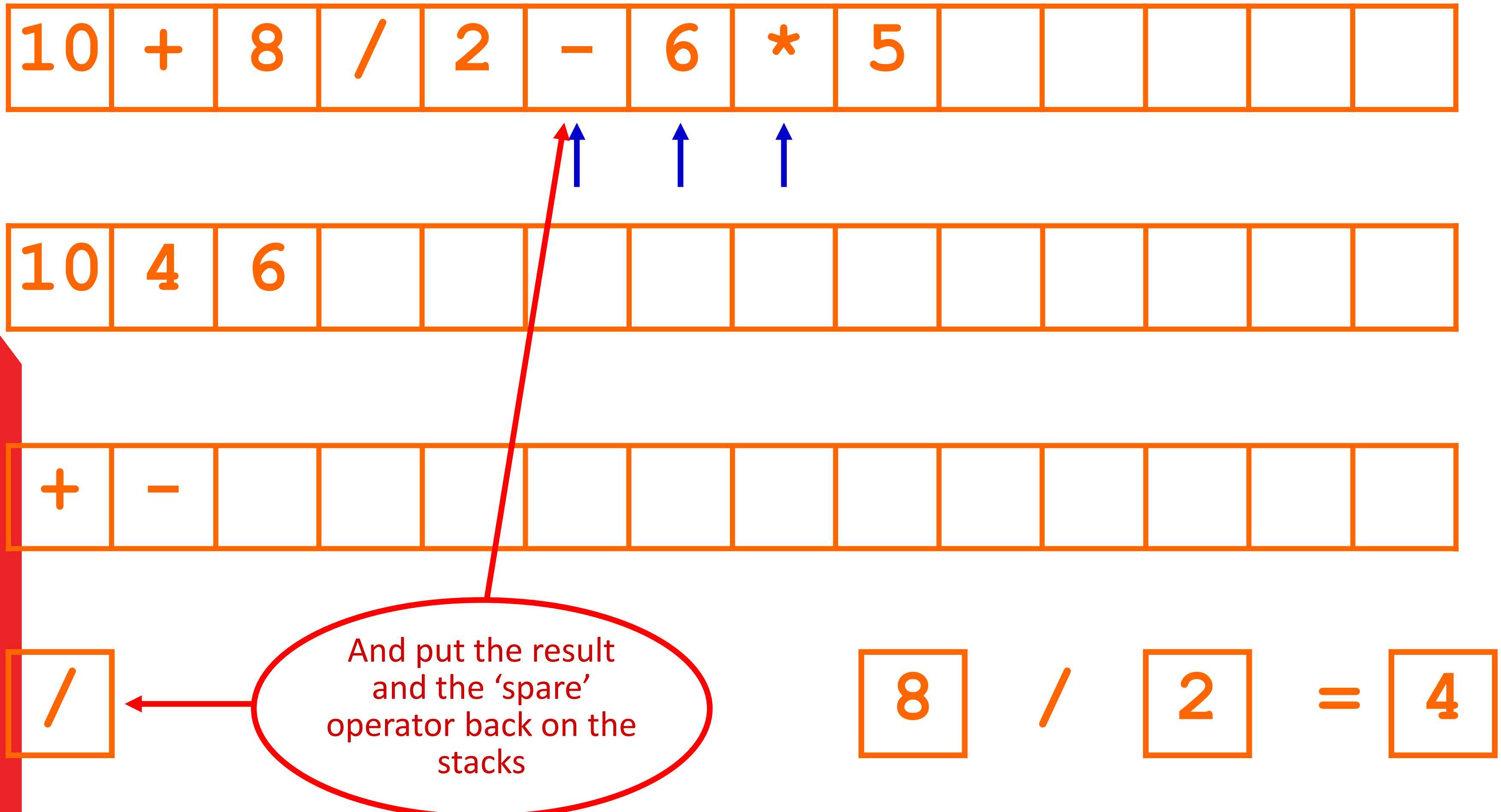
Stack Calculator Animation



Stack Calculator Animation



Stack Calculator Animation



Stack Calculator Animation

10	+	8	/	2	-	6	*	5					

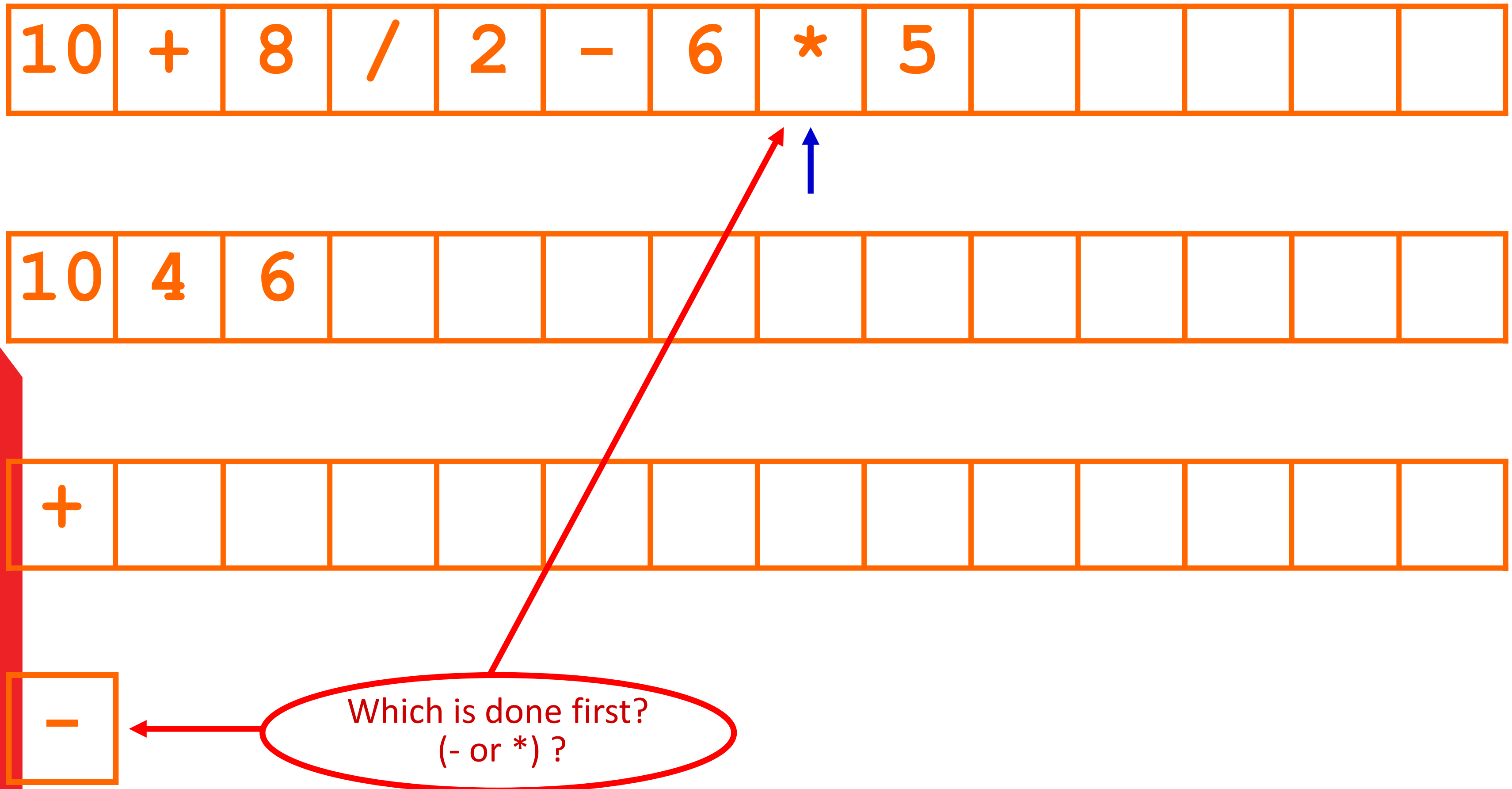
10	4	6											

+													

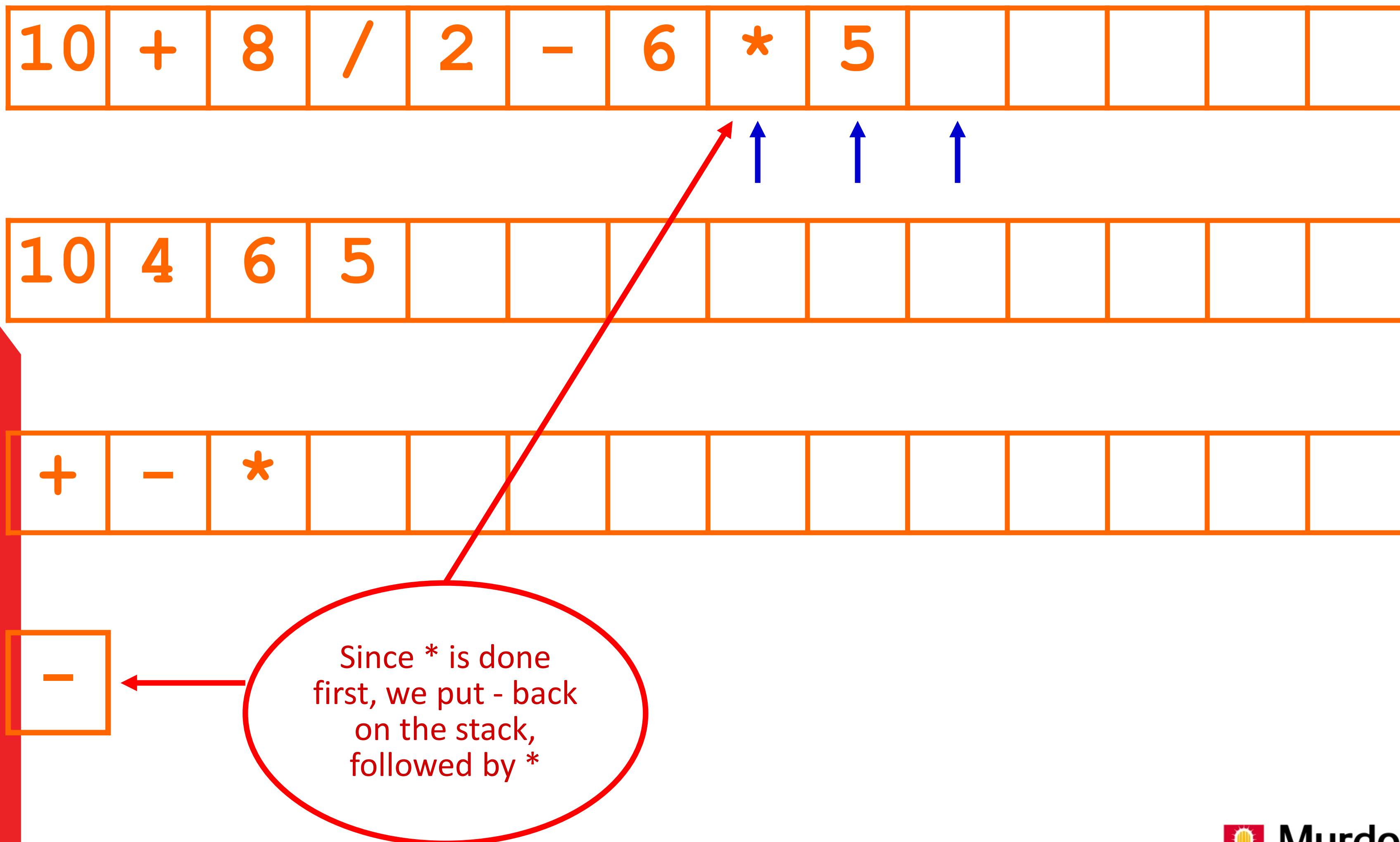
-

It is an operator, so pop the
last operator of the stack, for
comparison

Stack Calculator Animation



Stack Calculator Animation



Stack Calculator Animation

10	+	8	/	2	-	6	*	5					

10	4	6	5										

+	-	*											

*

We have reached the end of the equation, so we simply pop operations and numbers until finished

6 * 5 = 30

Murdoch
UNIVERSITY

Stack Calculator Animation

10	+	8	/	2	-	6	*	5					

10	4	30											

+	-												

*

$\boxed{6} * \boxed{5} = \boxed{30}$

Stack Calculator Animation

10	+	8	/	2	-	6	*	5					

10	4	30											

+	-												

$\boxed{-}$

$\boxed{4} - \boxed{30} = \boxed{-26}$

Stack Calculator Animation

10	+	8	/	2	-	6	*	5					

10	-26												

+													

+

$\boxed{10} + \boxed{-26} = \boxed{-16}$

Stack Calculator Animation

10	+	8	/	2	-	6	*	5					

-16													

+ -26 = -16

There are no more operations, so if there is one number on the other stack, we have our answer.
More than one number would give an error.

Murdoch UNIVERSITY

Data Structures and Abstractions

Stack Example

Lecture 23

A Calculator

It is possible to use two stacks to do simple one line calculations.

- The first stack stores operators (characters) that have not yet been performed.
- We will start with just + - * /.
- The second stack stores the numbers being operated upon.
- Therefore we are trying to find the answer to something like

  10 + 8 / 2 – 6 * 5
- What *is* the answer to this?
- For simplicity's sake, we will assume integer input, but floating point output.

Using Diagrams

- Try this yourself or in a group.
- Draw up an array (set of boxes) to represent the string:

  "10 + 8 / 2 – 6 * 5"

  With one number (not digit but the whole integer) or operation in each box
- Draw an array for the number stack and one for the operation stack
- Figure how to do it with diagrams first.

Test Data

- The next thing, of course, is to design the test data: build the test plan.
- Construction of the test plan occurs **before** any code is written.
  - The test plan is written once you have analysed the problem to be solved
  - Gets added to as software development progresses.
- I came up with over 50 possibilities that should be tested in the test plan!
  - See the spreadsheet with testdata related to this lecture note.
  - More extended examples of testing in the *"testing 4 later units"* folder.
  - You must perform regression testing. This can be "painful" so think of ways to automate the testing process. You don't have to use it in this unit but should in later units. Various approaches are used in industry.
- Ignore advice about test plans and testing at your own peril.

Top Level Algorithm

- We are used to the infix notation: 2 + 3 and using this notation means that when you want to override operator precedence, you need to use ( ) as in (2 + 3) * 5.
- In Polish (discovered by a Polish logican Jan Lukasiewicz) notation (prefix notation), there is no need to use ( ). Prefix notation: + 2 3
- Someone else came along later with something called **R**everse **P**olish **N**otation (postfix form). Postfix notation: 2 3 +
- With RPN, there is an additional advantage in that operators are in the correct order for digital computers.
- RPN examples: 2 3 + 5 *
- So think this way: push the numbers on the stack until you get an operator. Then pop the last two numbers of the stack and apply the operator. Put the result back on the stack and repeat the whole process. This is easy. The question is how to convert from infix to RPN (postfix). [1]

Top Level Algorithm

- Assuming that we have the equation in a string, try to design an algorithm that will do the top level of process control of the string....
- Use what you understood when you tried to figure it out using a diagram. If you have forgotten go through using diagrams again.
- In other words, most of it will be enclosed in a loop

  WHILE more characters

  ENDWHILE
- Remember to keep it a *control* function: put off until later working out how things are actually done.
- In other words, concentrate on *what* not *how.*
- Normally (of course), we would be designing, coding and testing in parallel.

Next...

- Next work on each part of the algorithm that you have got, as a *what* not a *how.*
- Ideally, you should do this with a group of two or three other people. But you can always give it a go on your own.
- Run the Animation PowerPoint to see usage of a Stack Calculator

Readings

- Textbook: Stacks and Queues, section on Application of Stacks: Postfix Expressions Calculator.
  - The RPN calculator is described in the above section.
  - There are a number of calculators which accept RPN entry and therefore make calculations of long expressions easy – less keys to press to get the same result.
    - Your mobile phone may have a RPN option.

Murdoch UNIVERSITY

Murdoch University

Searching, Merging and Sorting

Lecture 25

Murdoch University

Testing

- When testing searching and merging algorithms, it is important to check boundary and unusual conditions:
- In other words, test for containers with **[1]**:
  - 0 elements;
  - 1 element;
  - 2 elements;
  - 3 elements;
  - a large number of odd elements;
  - a large number of even elements;
  - The Cyclomatic Complexity of your algorithm can be used to guide your test cases. **[2]**

Murdoch UNIVERSITY

Linear Search

- Linear searches involve starting at the beginning, then checking each element in the container until we find the right one.
- In other words, a brute force approach.
- This is sure but slow: its average complexity is O(n). [1]
- This code is usually put inside a Find() or Search() routine.
- It is the only search available to linked lists and unsorted arrays and is therefore the search used for the STL vector and list.

Murdoch UNIVERSITY

Linear Search Algorithm

```
Boolean Find (DataClass target, Address targetPosition) [code in
textbook]

     Boolean found
     Set found to false
     Start at the beginning of the container
     WHILE not at the end of container AND found is false
          IF the current element is the target
               targetPosition = Address (index of the current
                                         element}
               found = true
          ENDIF
          set current to next element
     ENDWHILE
     Return found

END FIND
```

Murdoch UNIVERSITY

Binary Search

- A faster search than linear search exists for **sorted**, direct access containers such as sets and maps.
- This is *binary* search, where the search space is halved after each guess.
- The "Guess a number between 1 and 100" game played by children is a binary search.
- It is a divide and conquer strategy.
- The order of complexity is O(log(n)) for the number of iterations.
- See diagrams explaining this in the textbook – section on binary search, Chapter on Searching and Sorting Algorithms.
  - Go through the worked examples in this chapter found in the section on Asymptotic Notation: Big-O Notation. **[1]**

Murdoch UNIVERSITY

Iterative Binary Search Algorithm

```
•    Find (DataClass target, integer targetIndex) [code in textbook, data must
     be sorted]

•         integer bottomIndex, middleIndex, topIndex  [1]
•         boolean targetFound
•         targetFound = false
•         bottomIndex = 0
•         topIndex = arraySize-1
•         WHILE topIndex >= bottomIndex AND targetFound = false
•              middleIndex = (topIndex + bottomIndex)/2
•              IF target = value at middleIndex
•                targetIndex = middleIndex
•                targetFound = true
•              ELSEIF target < value at middleIndex
•                topIndex = middleIndex-1 // no point searching above
•              ELSEIF target > value at middleIndex
•                bottomIndex = middleIndex+1 // no point searching below
•              ENDIF
•         ENDWHILE

•    END Find
```

Murdoch UNIVERSITY

Iteration versus Recursion

- Anything that can be done with recursion can be done with iteration.
- Anything that can be done with iteration can be done with recursion.
- Advantages of Iteration:
  - Often easier to understand
  - Uses less memory
- Advantages of Recursion: **[1]**
  - Sometimes much easier to understand
  - Often simpler to code
  - Reduces code complexity

Recursive Binary Search Algorithm [1]

```
•    Find (DataClass target, integer targetIndex) : boolean
•         Set targetIndex to -1
•         return Find (target, 0, arraySize-1, targetIndex)
•    End Find

•    Find (DataClass target, integer bottomIndex, integer topIndex,
•          integer targetIndex) : boolean // the overloaded version
•         Boolean found
•         Set found to false
•         Integer middleIndex
•         middleIndex = (topIndex + bottomIndex)/2
•         IF target = array[middleIndex]
•              targetIndex = middleIndex
•              found = true        // line A
•         ELSEIF topIndex <= bottomIndex
•              found = false
•         ELSEIF target < array[middleIndex]
•              Find (target, bottomIndex, middleIndex-1, targetIndex)//Line B
•         ELSEIF target > array[middleIndex]
•              Find (target, middleIndex+1, topIndex, targetIndex)
•         ENDIF
•         Return found [2]       // Line C
•    End Find
```

Merging Sorted Containers

- When we looked at Sets in a previous lecture, we looked at algorithms for subset, difference, union and intersection.
- They all operate in O(n) time.
- They were all very similar.
- This is because they were all variations of the standard *merge* algorithm for sorted containers.
- The merge algorithm is also important for *merge sort* which is the best (only) sort to use for very large amounts of data stored on disk.
- Note that the STL `<algorithm>` class contains a merge algorithm that works on **sorted** containers.

Murdoch UNIVERSITY

```
Merge(container1, container2, newContainer) [1]

      datum1 = first element in container1
      datum2 = first element in container2
      WHILE there are elements in both container1 and container2
            IF datum1 < datum2
                  Put datum1 in newContainer
                  datum1 = next element in container1
            ELSEIF datum2 < datum1
                  Put datum2 in newContainer
                  datum2 = next element in container2
            ELSE
                  Put datum1 in newContainer
                  Put datum2 in newContainer  // duplicates are being kept
                  datum1 = next element in container1
                  datum2 = next element in container2
            ENDIF
      ENDWHILE

      WHILE there are elements in container1
            Put datum1 in newContainer
            datum1 = next element in container1
      ENDWHILE

      WHILE there are elements in container2
            Put datum2 in newContainer
            datum2 = next element in container2
      ENDWHILE

End Merge
```

Categorisation of Sorting Algorithms

- Categorising sorting algorithms allows decisions to be made on the best sort to use in a particular situation.
- Algorithms are categorised based on:
  - what is actually moved (direct or indirect);
  - where the data is stored during the process (internal or external);
  - whether prior order is maintained (stable vs unstable);
  - how the sort progresses;
  - how many *comparisons* are made on average and in the worst case;
  - how many *moves* are made on average and in the worst case.

Murdoch UNIVERSITY

Direct vs Indirect

- Direct sorting involves moving the elements themselves.
  For example when sorting an array

50	20	10	60	10

It becomes

10	10	20	50	60

- Indirect sorting involves moving objects that designate the elements (also called address table sorting). This is particularly common where the actual data is stored on disk or in a database.
  For example, if sorting an array:

50	20	10	60	10

we do not sort the data, but instead set up an array of the addresses:

0	1	2	3	4

and sort them based on the data to which they refer:

2	4	1	0	3

Murdoch UNIVERSITY

Internal vs External

- Internal: the data is stored in RAM.
- External: the data is stored on secondary storage (hard drive, tape, floppy disk etc).
- There are two external sorts: natural merge and polyphase. The latter is somewhat complicated and it is usually used for large files. [1]
  - We wouldn't be looking at polyphase sort in this unit – read out of interest.

Murdoch UNIVERSITY

Stable vs Unstable

- Stable sorts preserve the prior order of elements where the new order has equal keys.
- For example, if you have sorted on name and then sort on address, people with the same address would still be sorted on name.
- On the whole stable sorts are slower.

Type of Progression

- *Insertion*: examine one element at a time and insert it into the structure in the proper order relative to all previously processed elements.
- *Exchange*: as long as there are still elements out of order, select two elements and exchange them if they are in the wrong order.
- *Selection*: as long as there are elements to be processed, find the next largest (or smallest) element and set it aside.
- *Enumeration*: each element is compared to all others and placed accordingly. [1]
- *Special Purpose*: a sort implemented for a particular one-off situation.

Murdoch UNIVERSITY

Number of Comparisons

Type	Name	Average O	Worst Case O
Insertion	Straight Insertion	n^2	n^2
	Binary Insertion	n log n	n log n
	Shell*	$n^{1.3}$	$n^{1.5}$
Exchange	Bubble	n^2	n^2
	Shaker	n^2	n^2
	Quicksort	n log n	n^2
	Merge	n log n	n log n
Selection	Straight Selection	n^2	n^2
	Heap	n log n	n log n

* Based on empirical evidence only.

Murdoch UNIVERSITY

Number of Comparisons [1]

Type	Name	Average O	Worst Case O
Insertion	Straight Insertion	n^2	n^2
	Binary Insertion	n log n	n log n
	Shell*	$n^{1.3}$	$n^{1.5}$
Exchange	Bubble	n^2	n^2
	Shaker	n^2	n^2
	Quicksort	n log n	n^2
	Merge	n log n	n log n
Selection	Straight Selection	n^2	n^2
	Heap	n log n	n log n

* Based on empirical evidence.

Murdoch UNIVERSITY

Number of Moves

Type	Name	Average O	Worst Case O
Insertion	Straight Insertion	n^2	n^2
	Binary Insertion	n^2	n^2
	Shell*	$n^{1.25}$	-
Exchange	Bubble	n^2	n^2
	Shaker	n^2	n^2
	Quicksort	n log n	n^2
	Merge	n log n	n^2
Selection	Straight Selection	n log n	n^2
	Heap	n log n	n log n

* Based on empirical evidence only

Murdoch UNIVERSITY

Number of Moves

Type	Name	Average O	Worst Case O
Insertion	Straight Insertion	n^2	n^2
	Binary Insertion	n^2	n^2
	Shell*	$n^{1.25}$	-
Exchange	Bubble	n^2	n^2
	Shaker	n^2	n^2
	Quicksort	n log n	n^2
	Merge	n log n	n^2
Selection	Straight Selection	n log n	n^2
	Heap	n log n	n log n

* Based on empirical evidence only

Murdoch UNIVERSITY

Algorithm Choice

- Looking at the tables, 'clearly' heap sort is the fastest, followed by mergesort and quicksort.
- So why is quicksort the algorithm used by spreadsheets, the STL, in C etc??
- There can be several reasons:
  - The first is that quicksort is an *internal* sort and the other two are *external* sorts. Therefore it requires less I/O, but there are versions of merge sort which try to cut down on I/O.
  - Obtaining and releasing memory is time consuming.
  - The next reason hidden in the use of big O notation. When running quicksort, merge and heap sort on my PC, I found that although they are all O(n log n) for random data, quicksort ran twice as fast as heap sort and almost 5 times faster than merge sort!
  - There are lots of very complicated ways to optimise quicksort.
  - On the flip side, merge sort is very suited to parallel programming.

Murdoch
UNIVERSITY

Readings

- Textbook Chapter Searching and Sorting Algorithms.
- Reference book, Introduction to Algorithms. For further study, see part of the book called Sorting and Order Statistics. It contains a number of chapters on sorting.

Murdoch UNIVERSITY

Data Structures and Abstractions

Sorting Algorithms

animations of algorithms

Lecture 26

Mur
UNIV

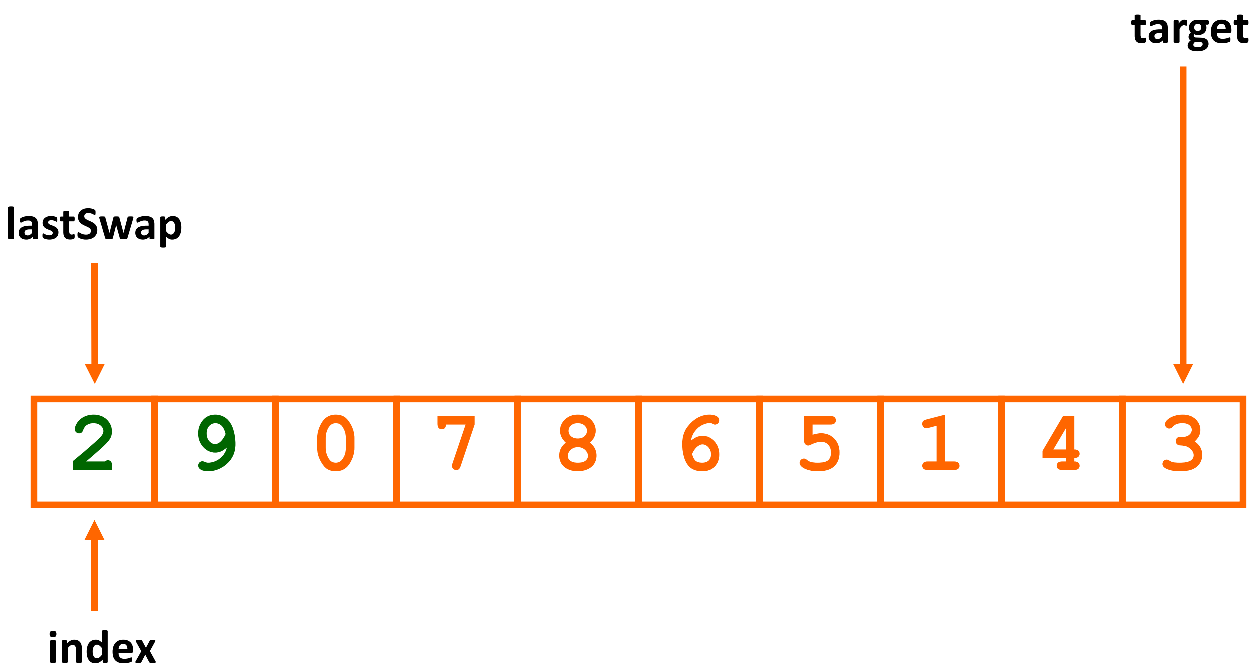
Bubble Sort

- Bubble sort is the most commonly coded of the simple sorts.
- It is a stable exchange sort.
- Whilst not particularly fast—$O(n^2)$—it is very simple to code and easy to understand.
- For anything less than 1000 items, bubble sort is fine.
- Its name derives from the fact that large numbers 'bubble' to the 'top' of the container.

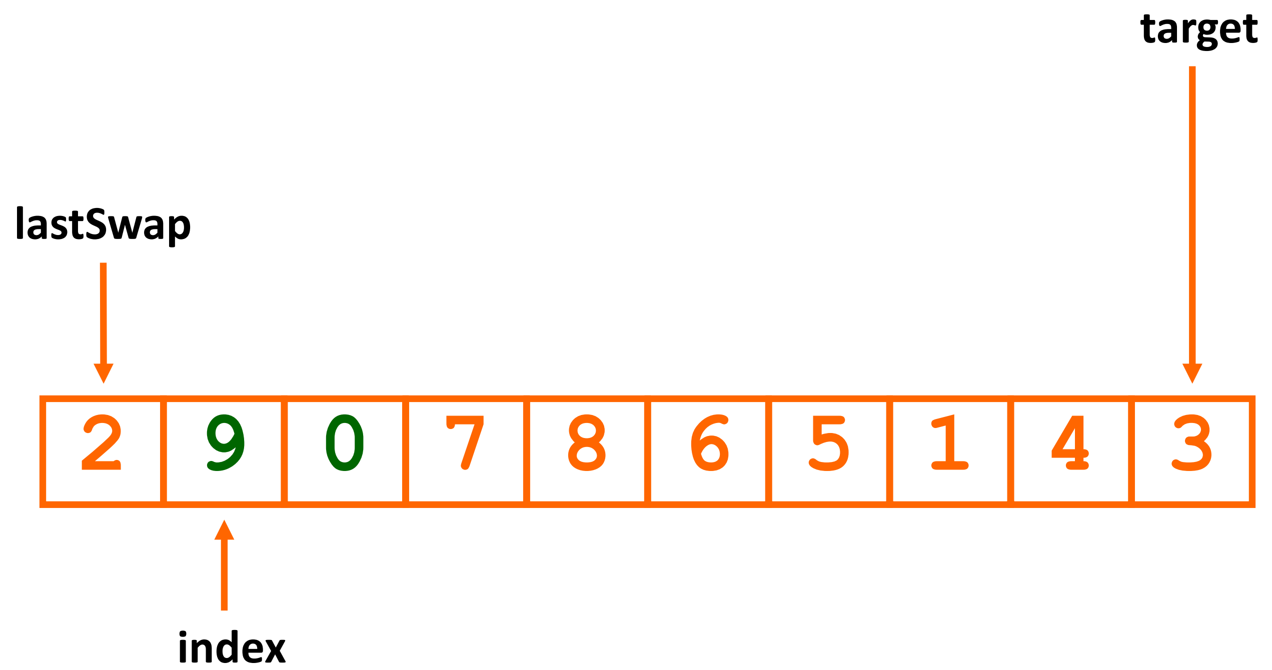
Bubble Sort Algorithm

```
•    ArrayBubbleSort

•         integer target, lastSwap
•         boolean swapDone, sortDone

•         Initialise lastSwap to 0
•         Initialise sortDone to false

•         IF array size > 1
•              target = size-1
•              WHILE not sortDone
•                   swapDone = false
•                   FOR index = 0 to target-1
•                        IF element[index] > element[index+1]
•                         Swap them
•                         lastSwap = index
•                         swapDone = true
•                        ENDIF
•                   ENDFOR
•                   sortDone = not swapDone
•                   target = lastSwap
•              ENDWHILE
•         ENDIF
•
•    END BubbleSort
```

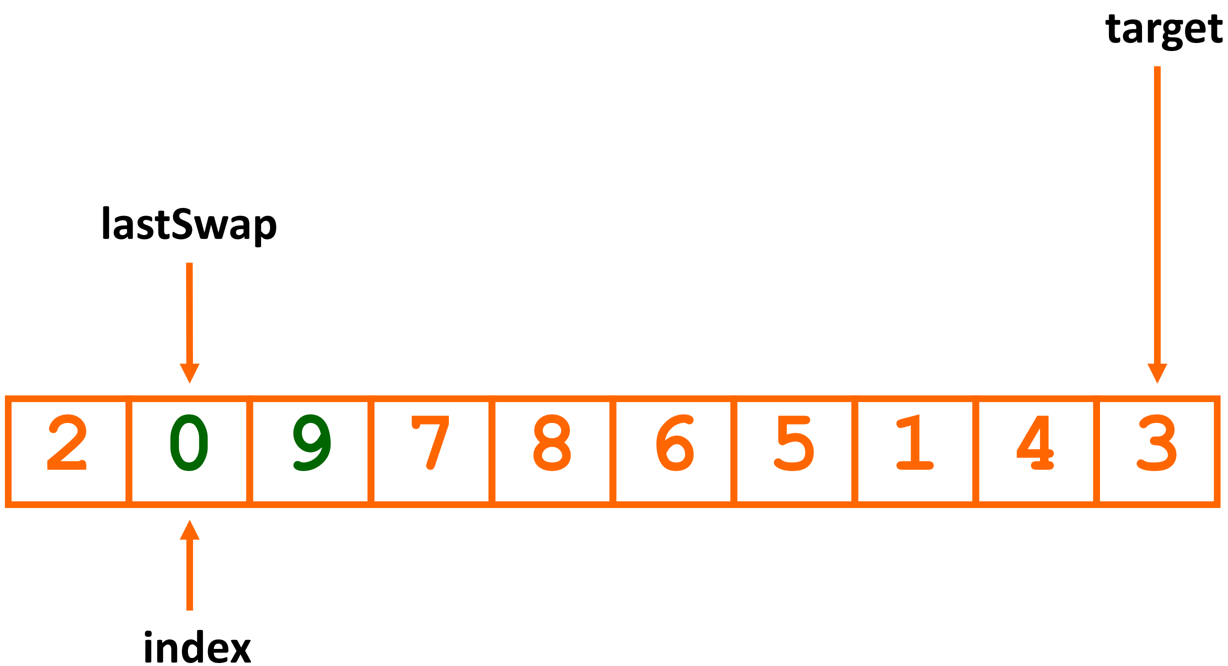
Bubble Sort Animation



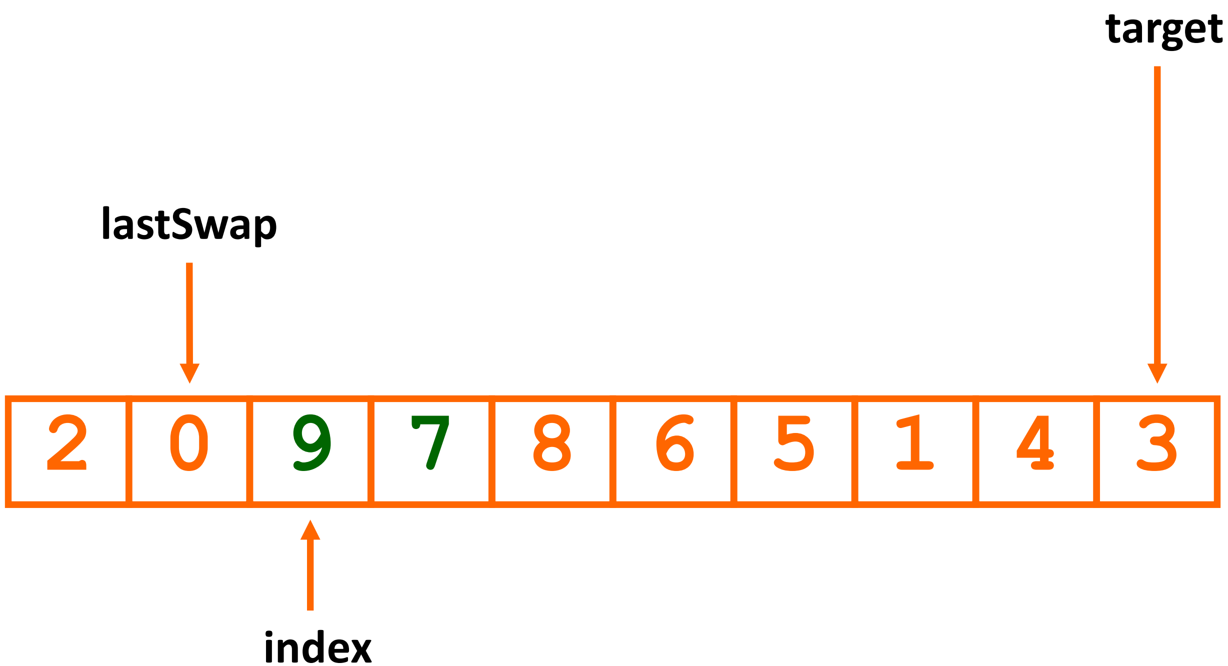
Bubble Sort Animation



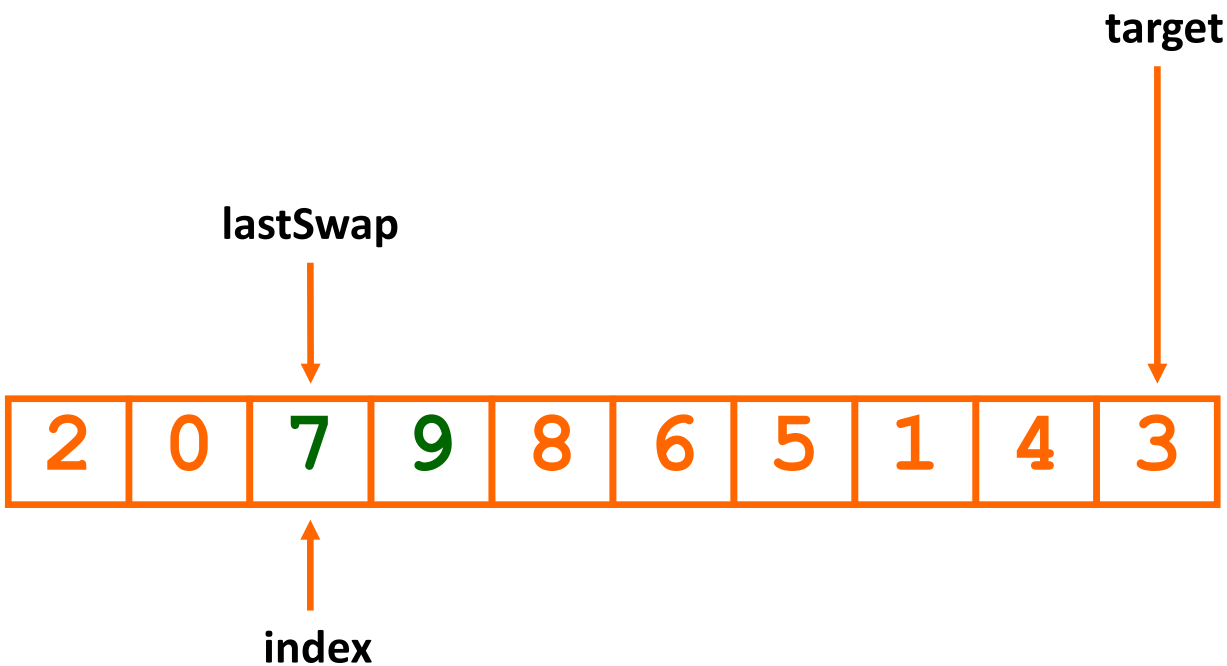
Bubble Sort Animation



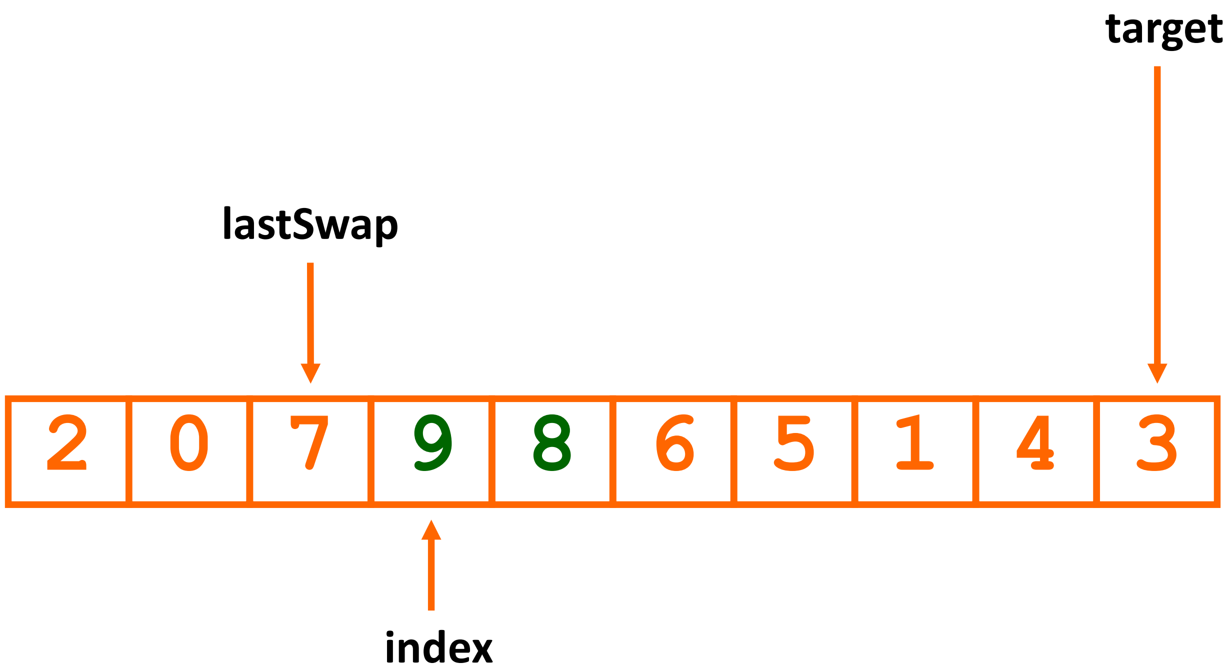
Bubble Sort Animation



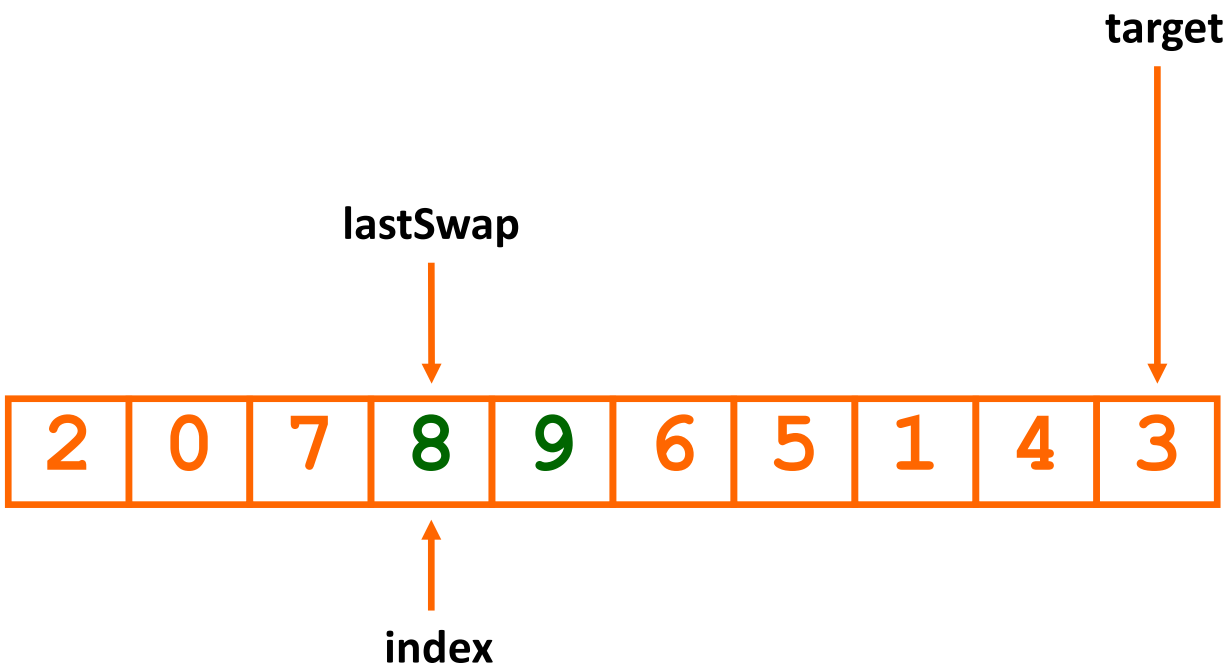
Bubble Sort Animation



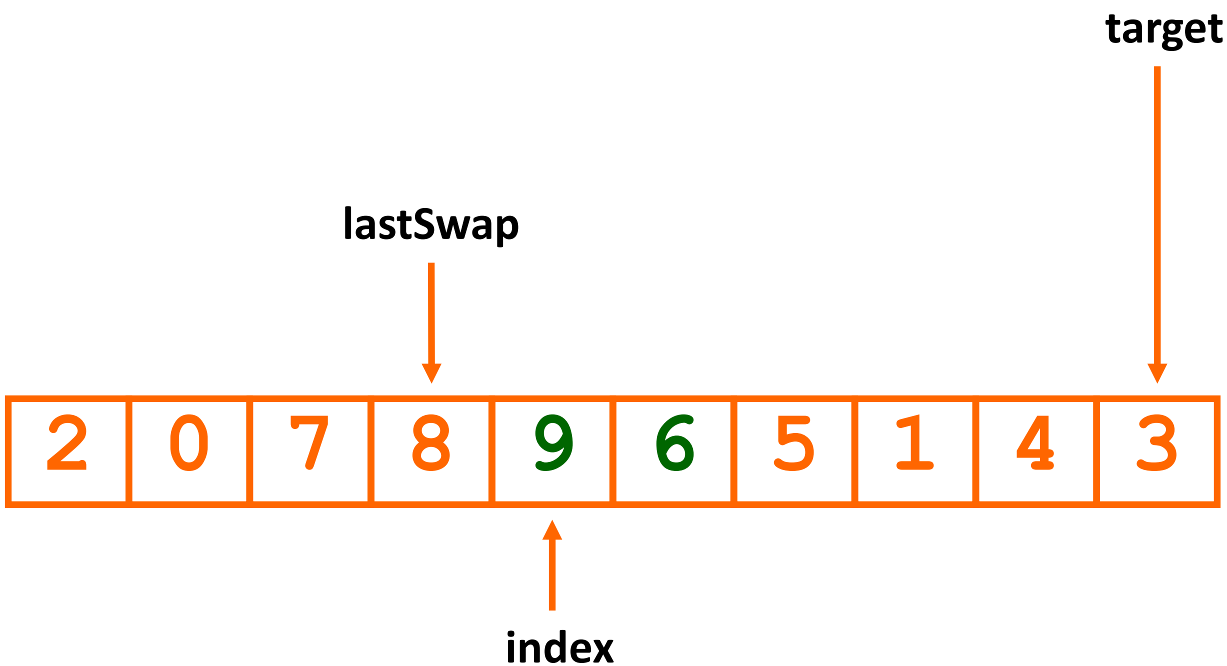
Bubble Sort Animation



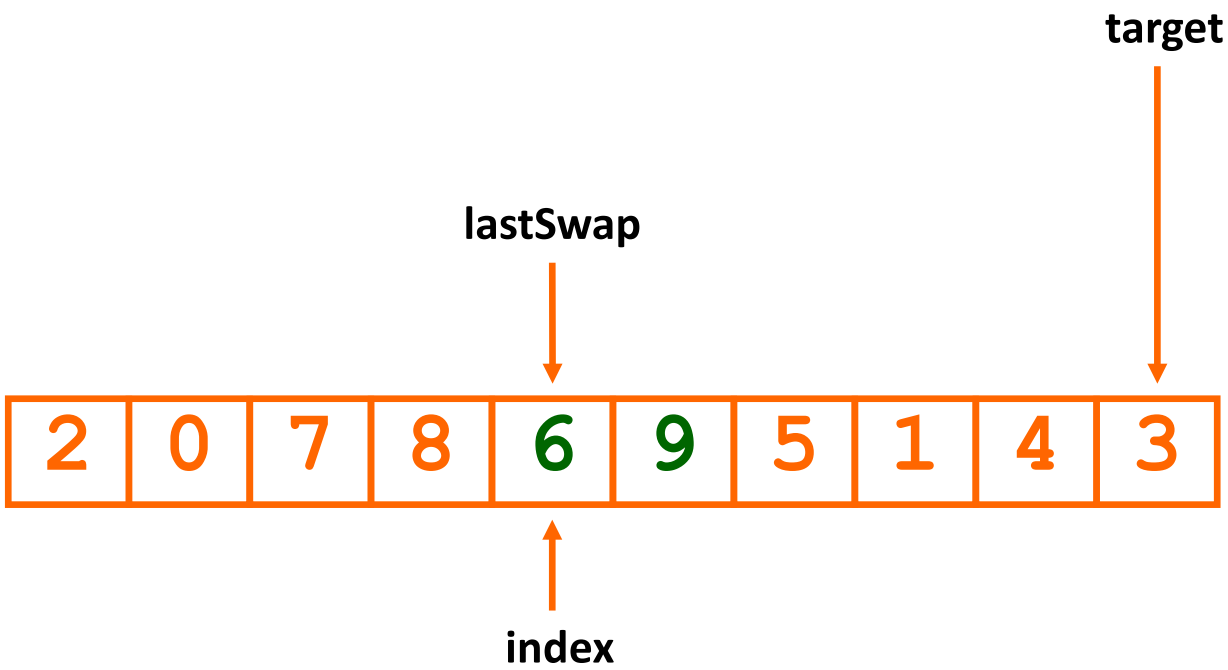
Bubble Sort Animation



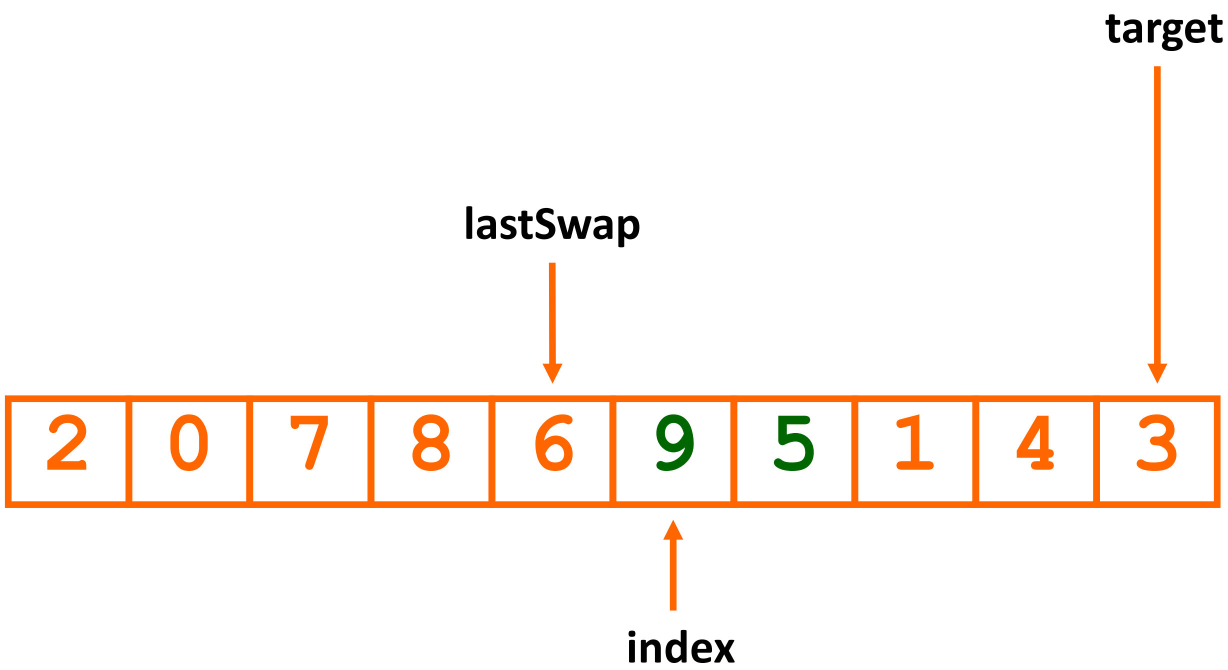
Bubble Sort Animation



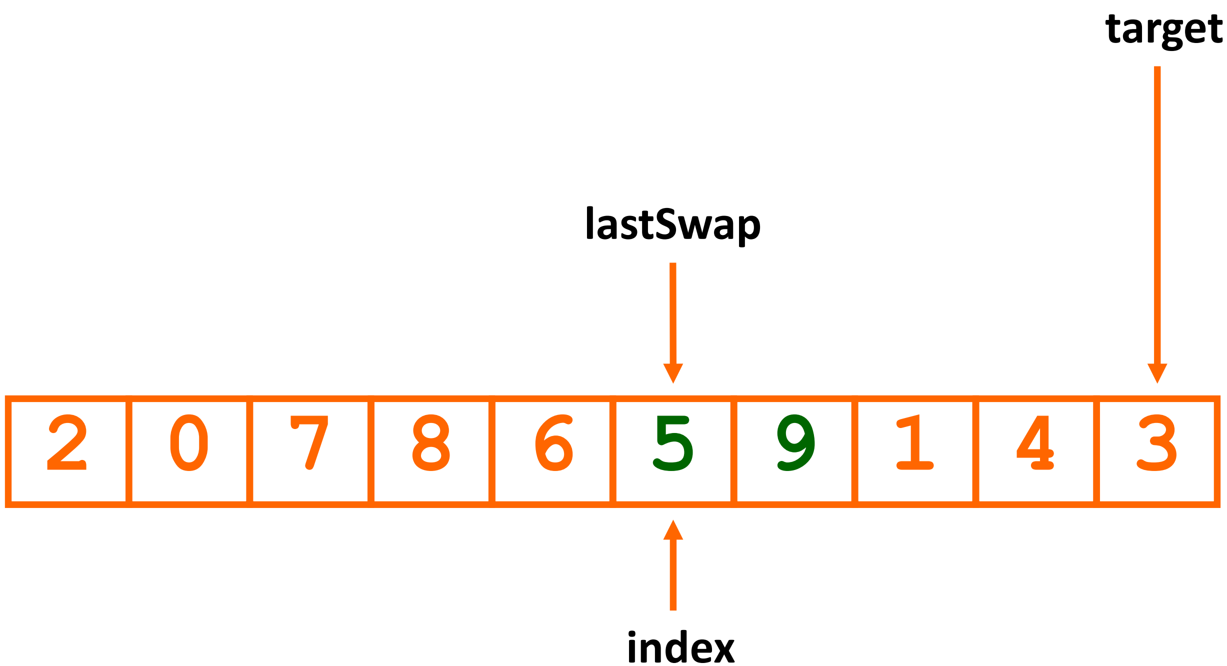
Bubble Sort Animation



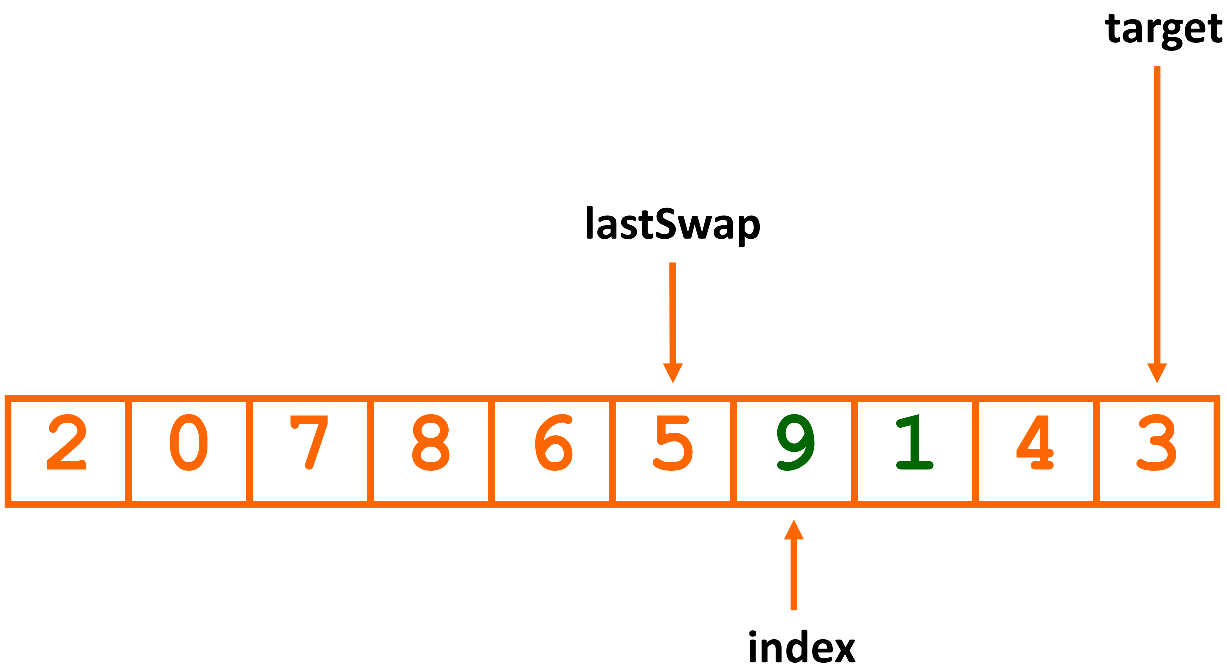
Bubble Sort Animation



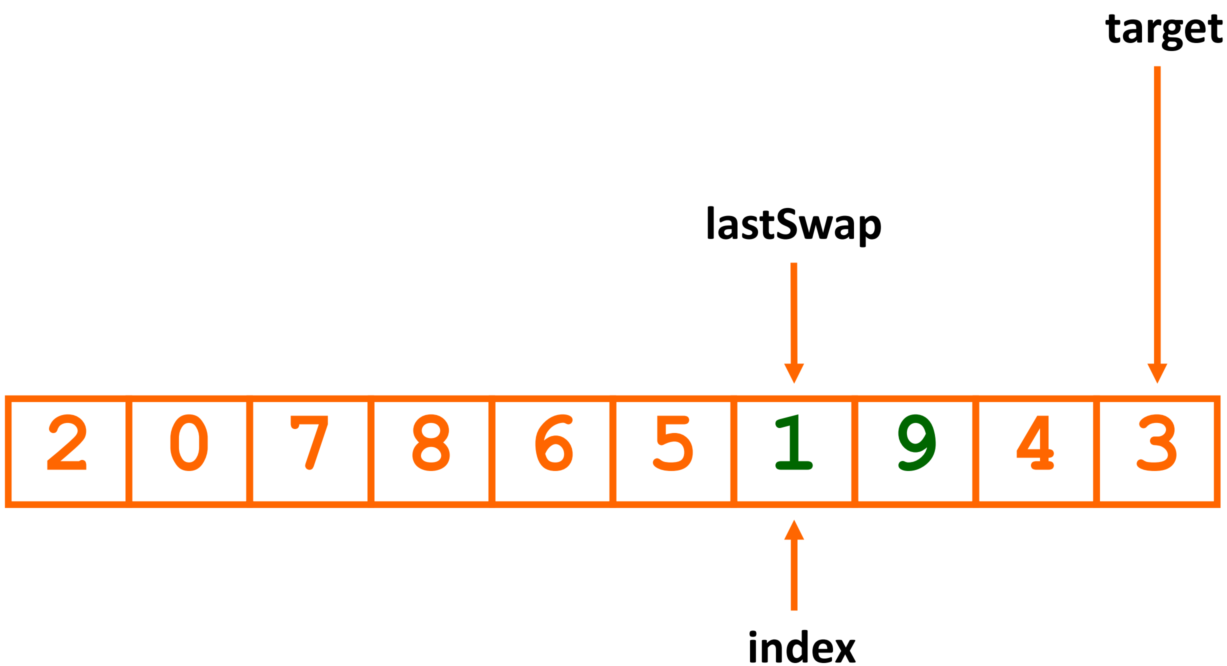
Bubble Sort Animation



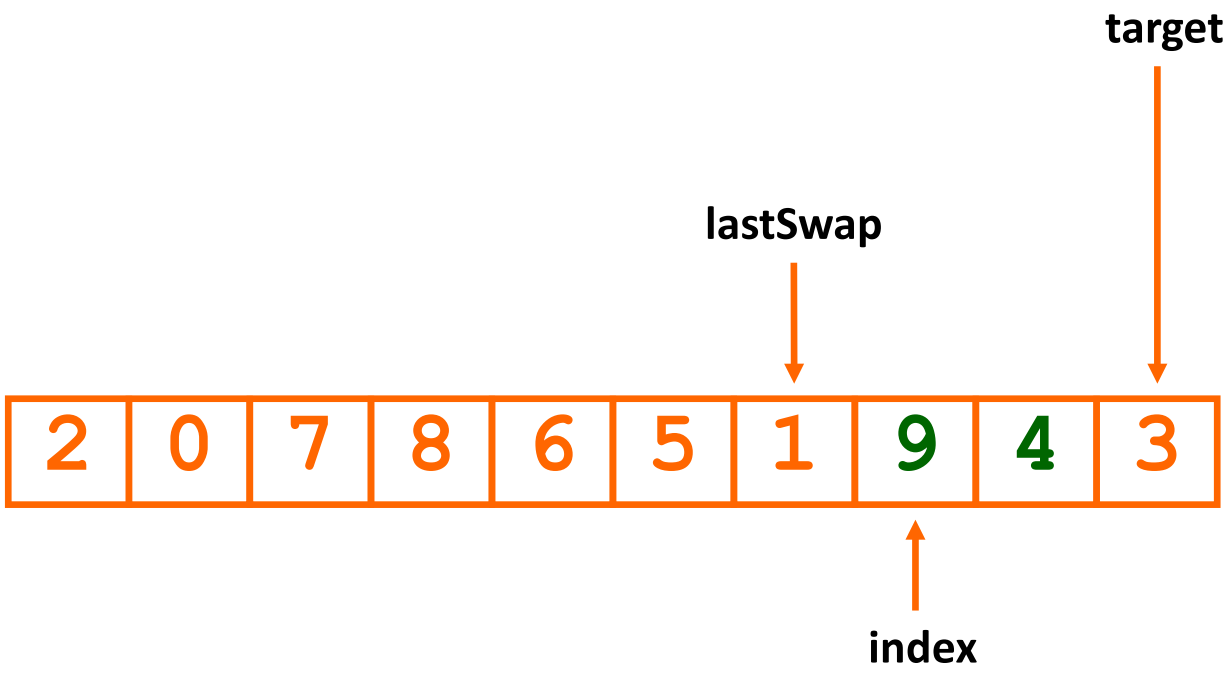
Bubble Sort Animation



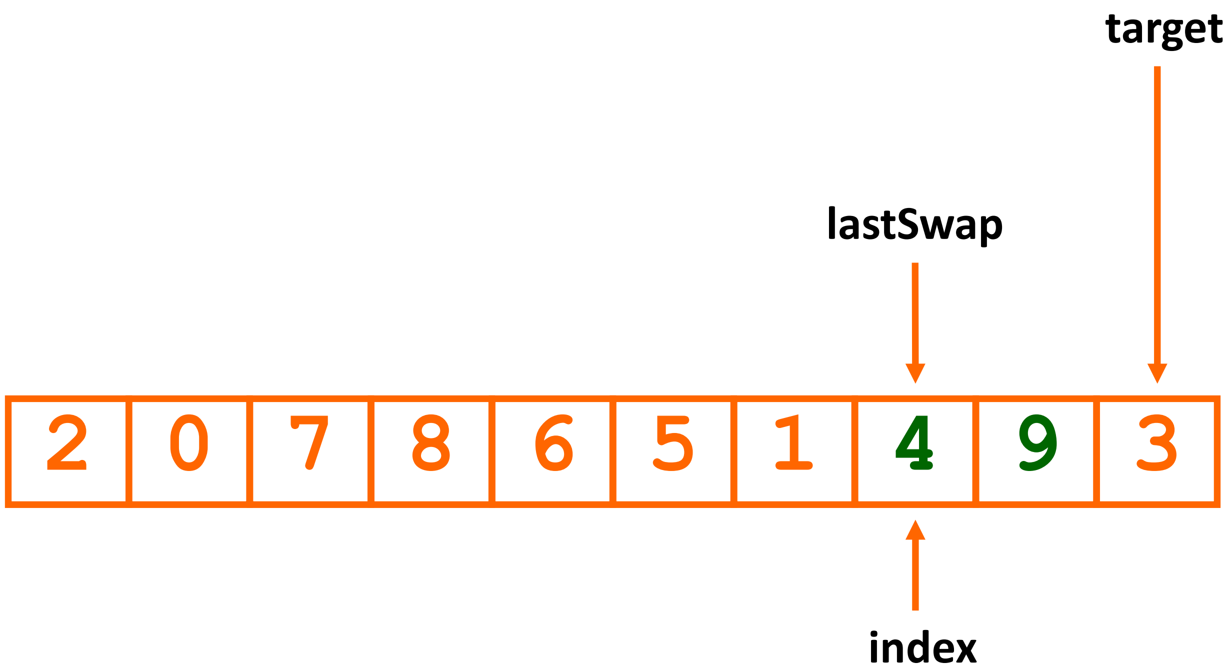
Bubble Sort Animation



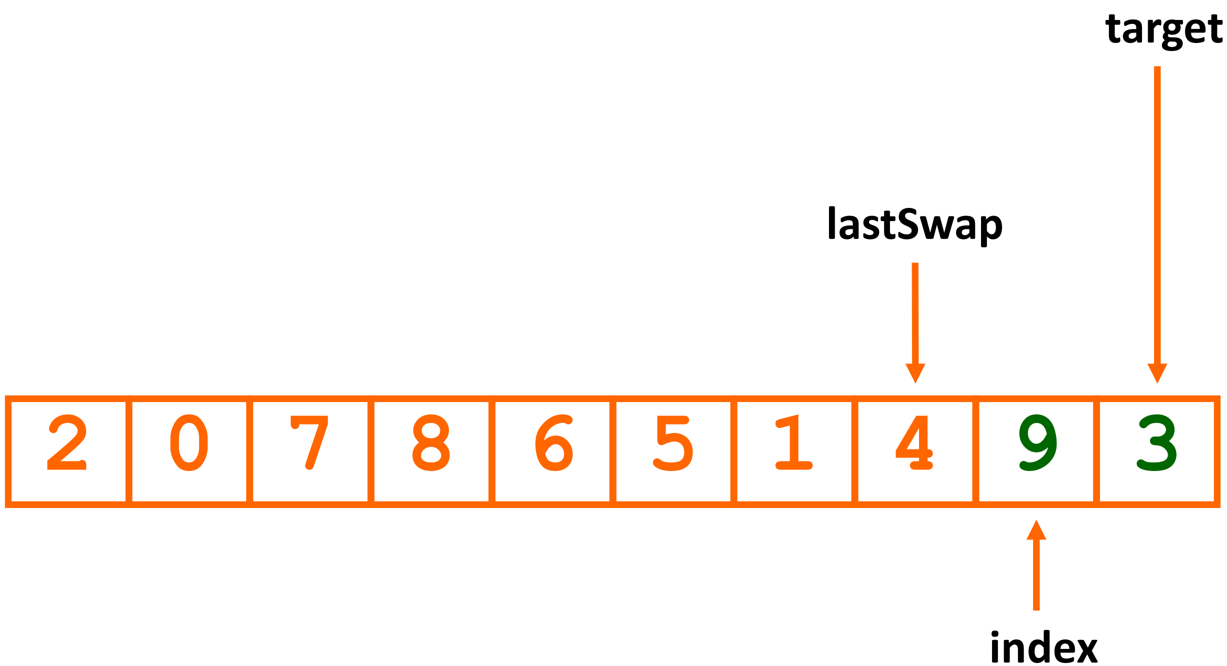
Bubble Sort Animation



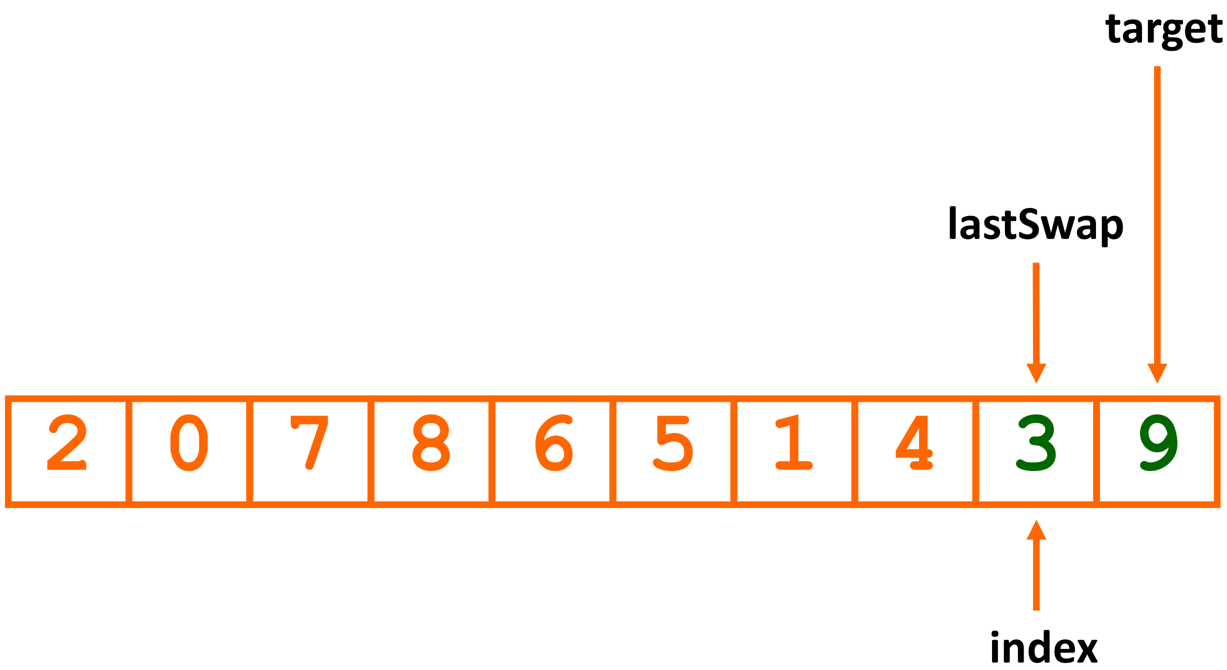
Bubble Sort Animation



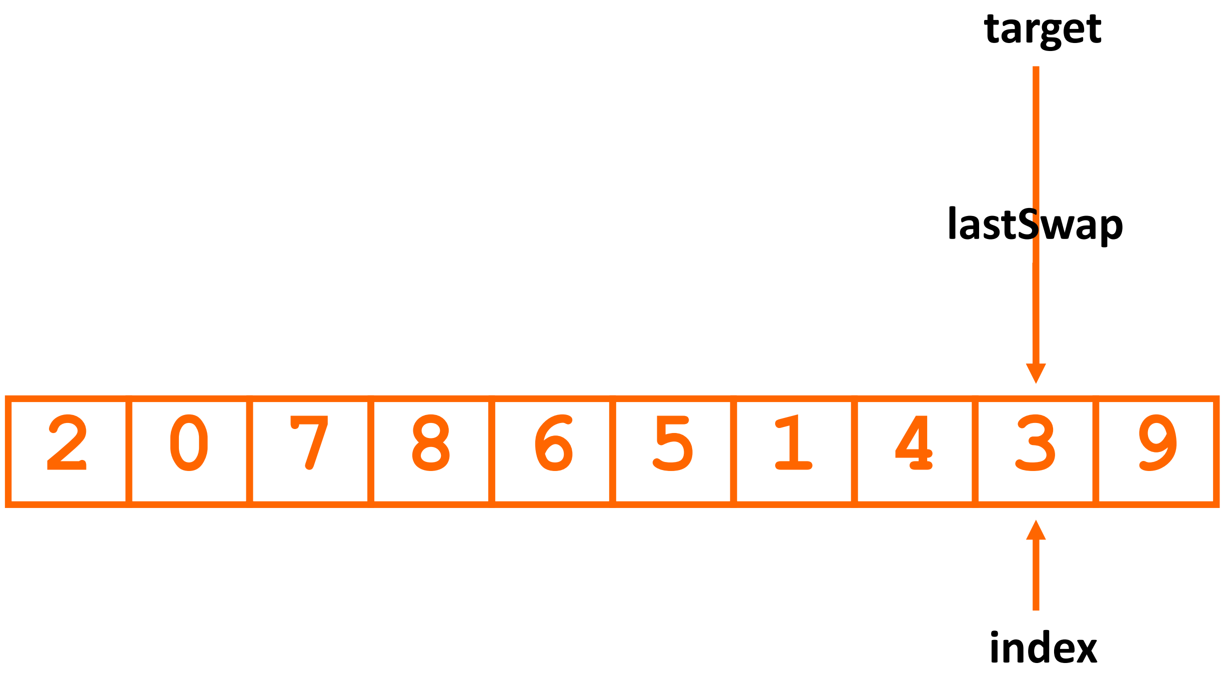
Bubble Sort Animation



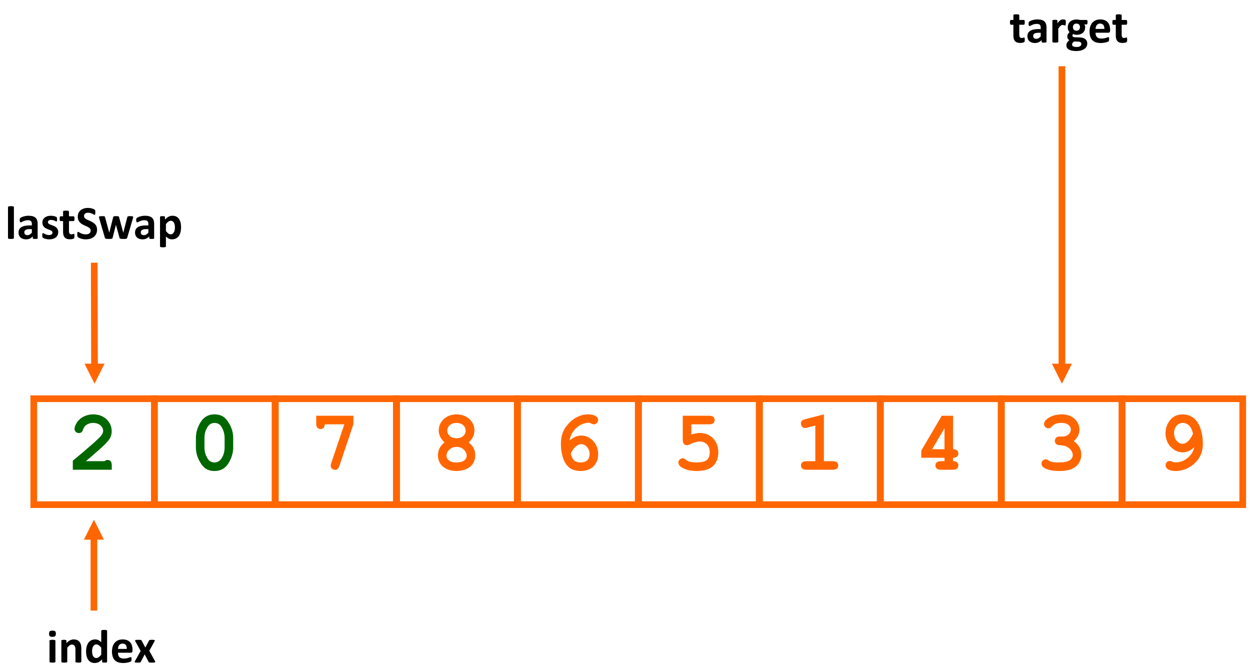
Bubble Sort Animation



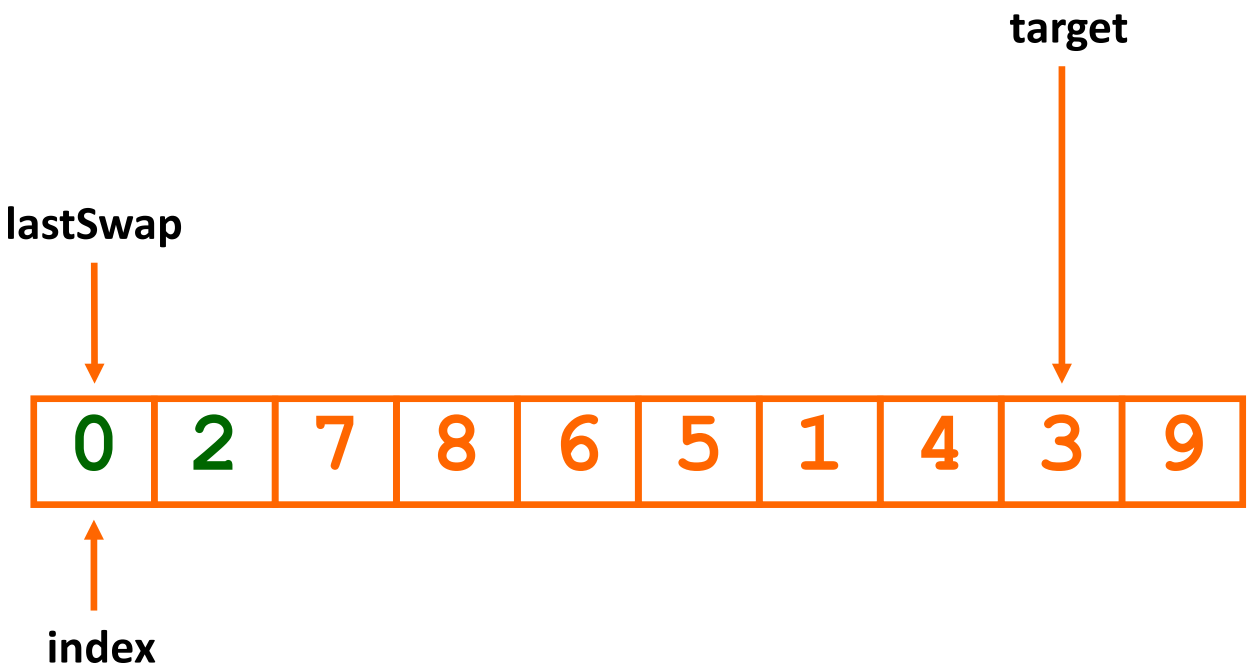
Bubble Sort Animation



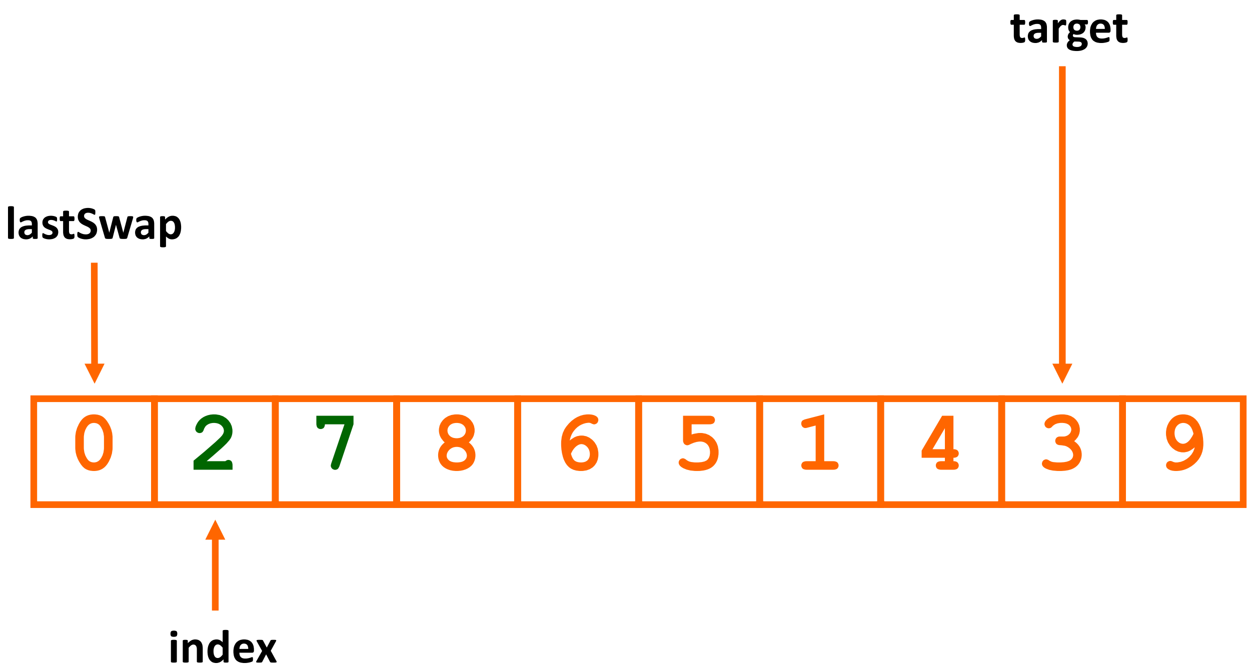
Bubble Sort Animation



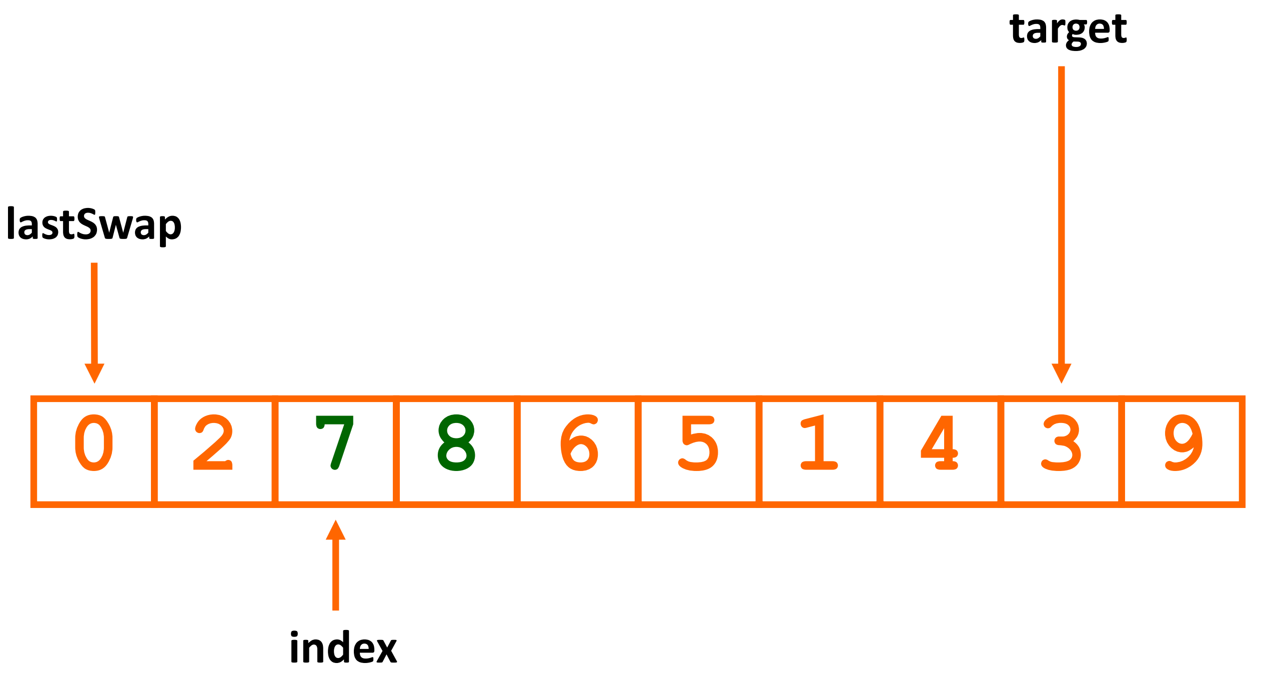
Bubble Sort Animation



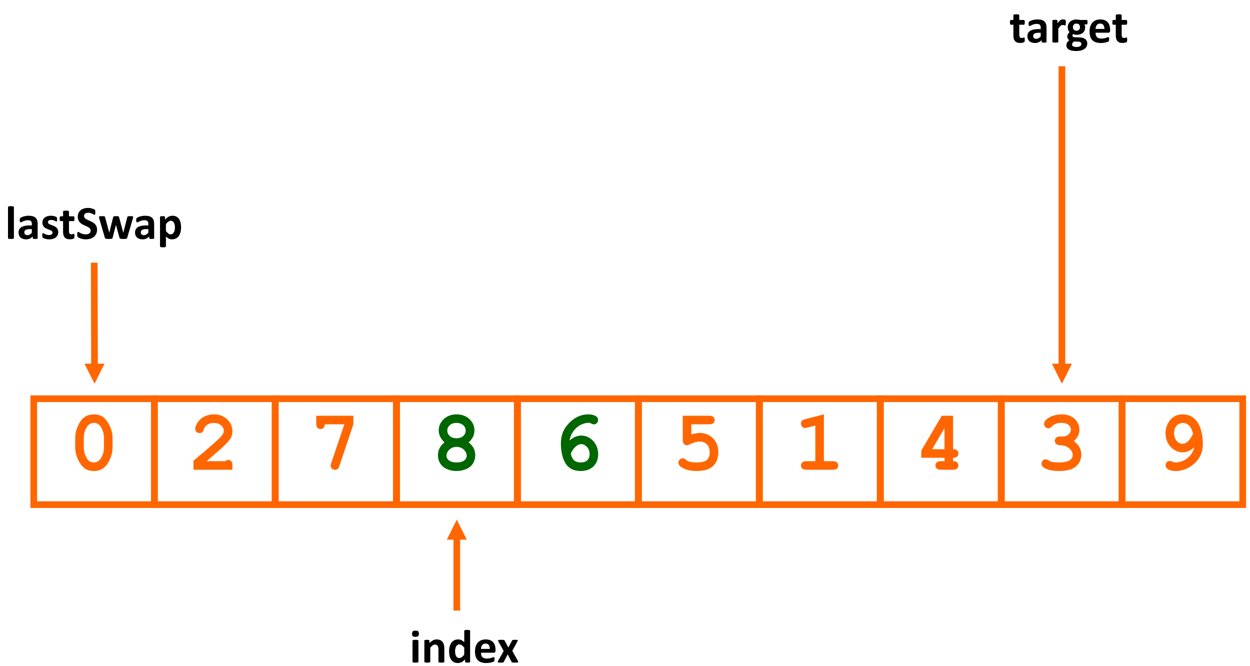
Bubble Sort Animation



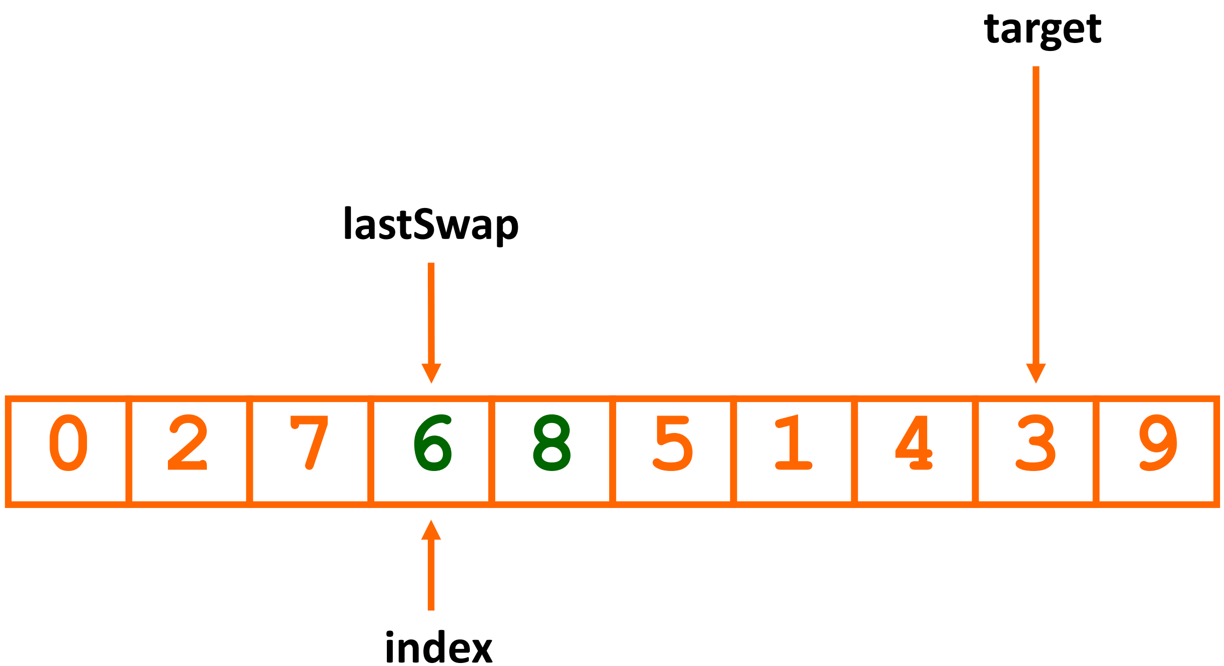
Bubble Sort Animation



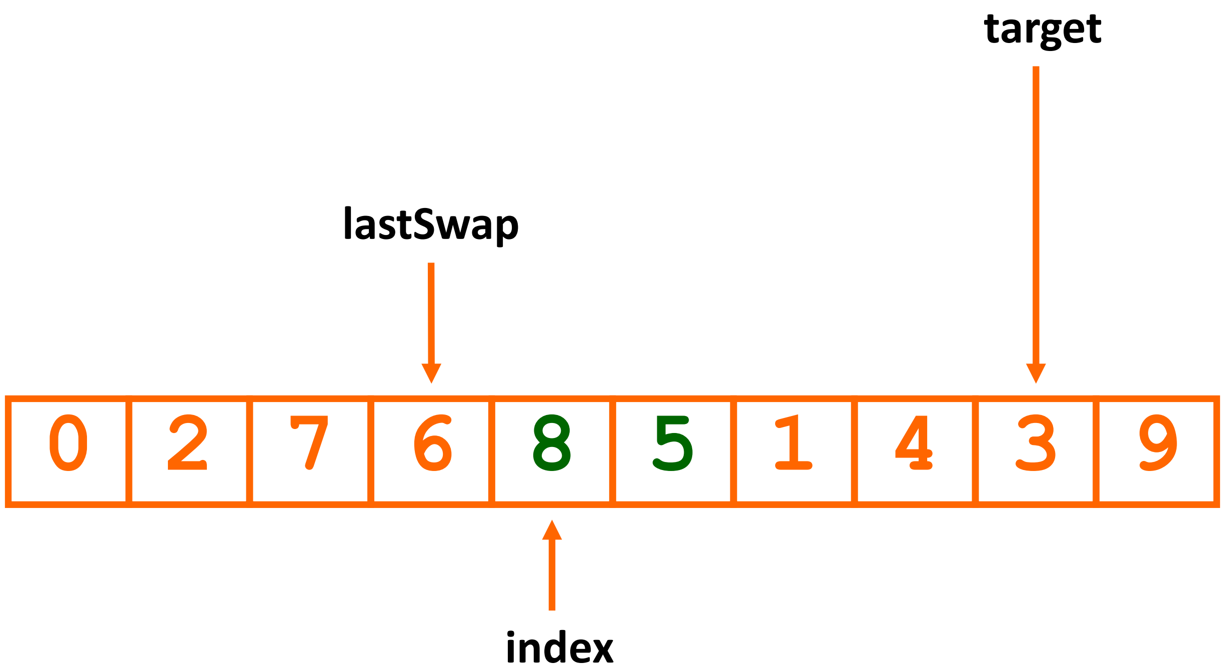
Bubble Sort Animation



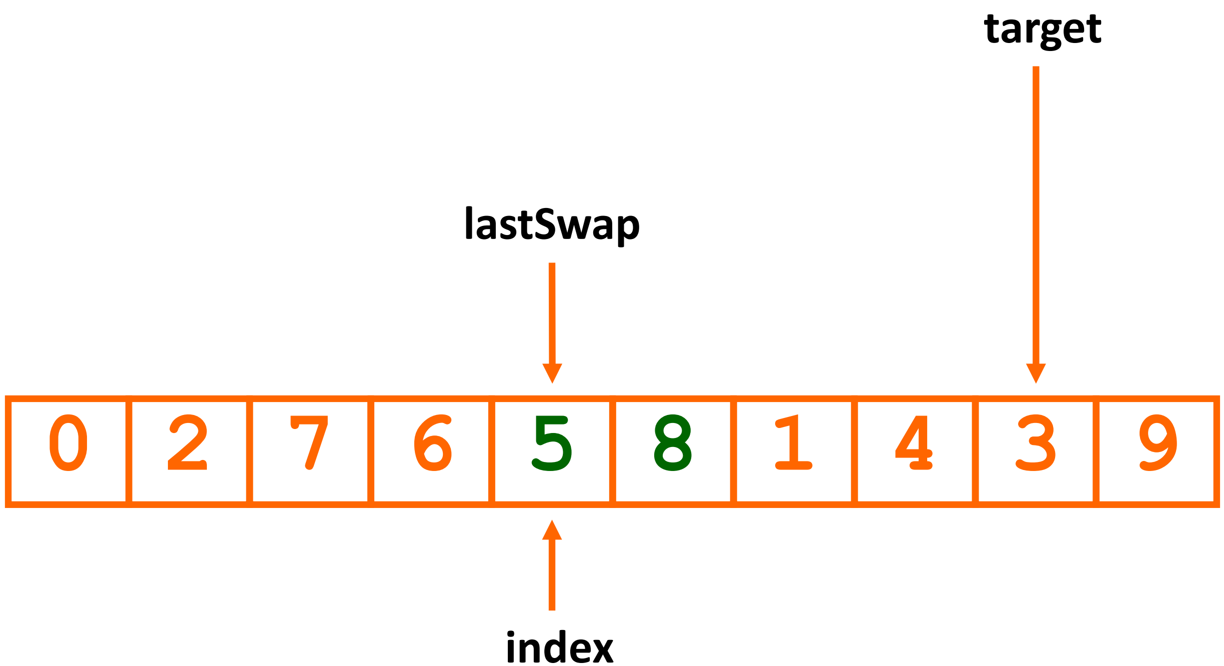
Bubble Sort Animation



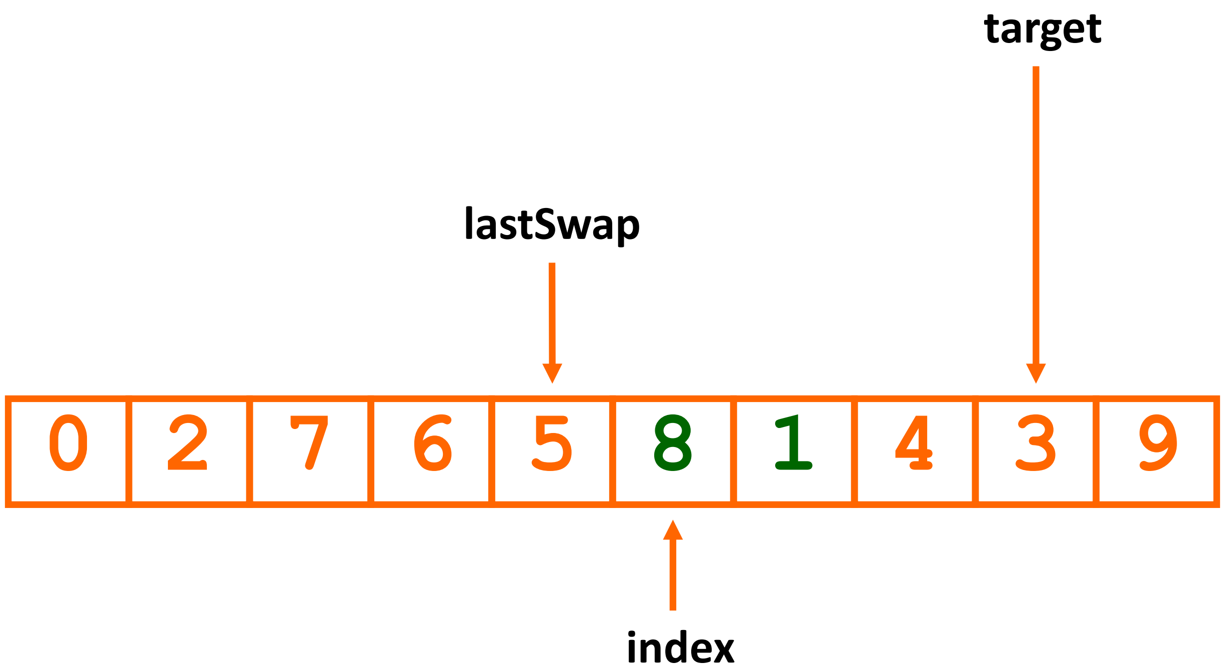
Bubble Sort Animation



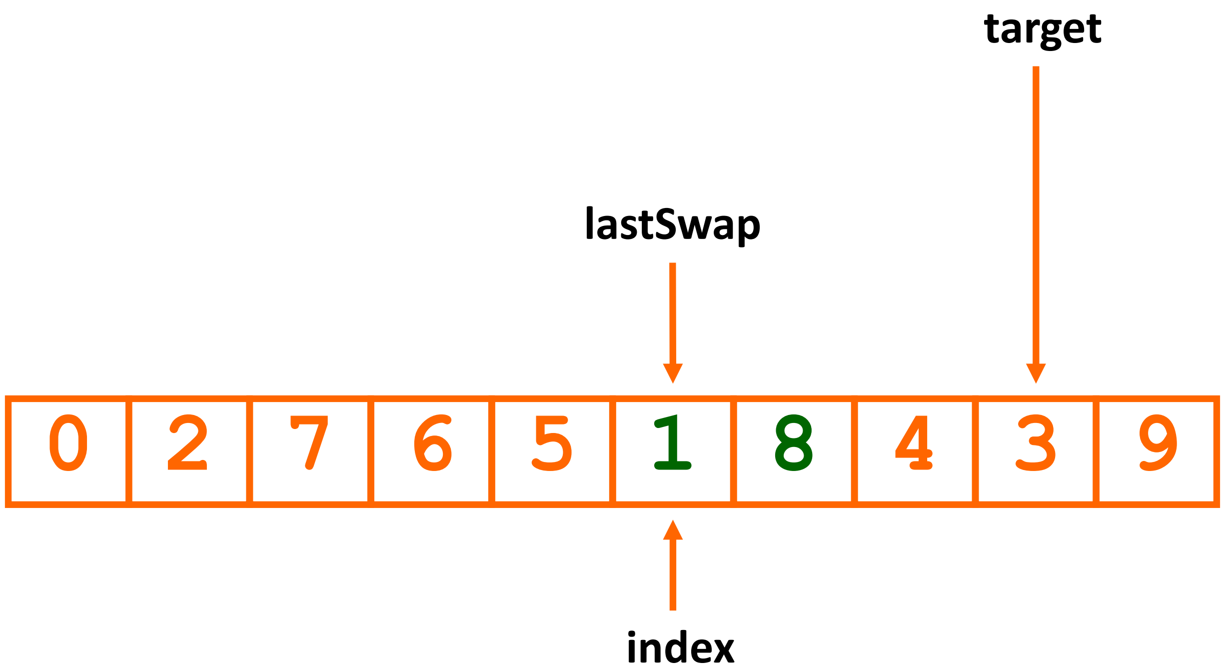
Bubble Sort Animation



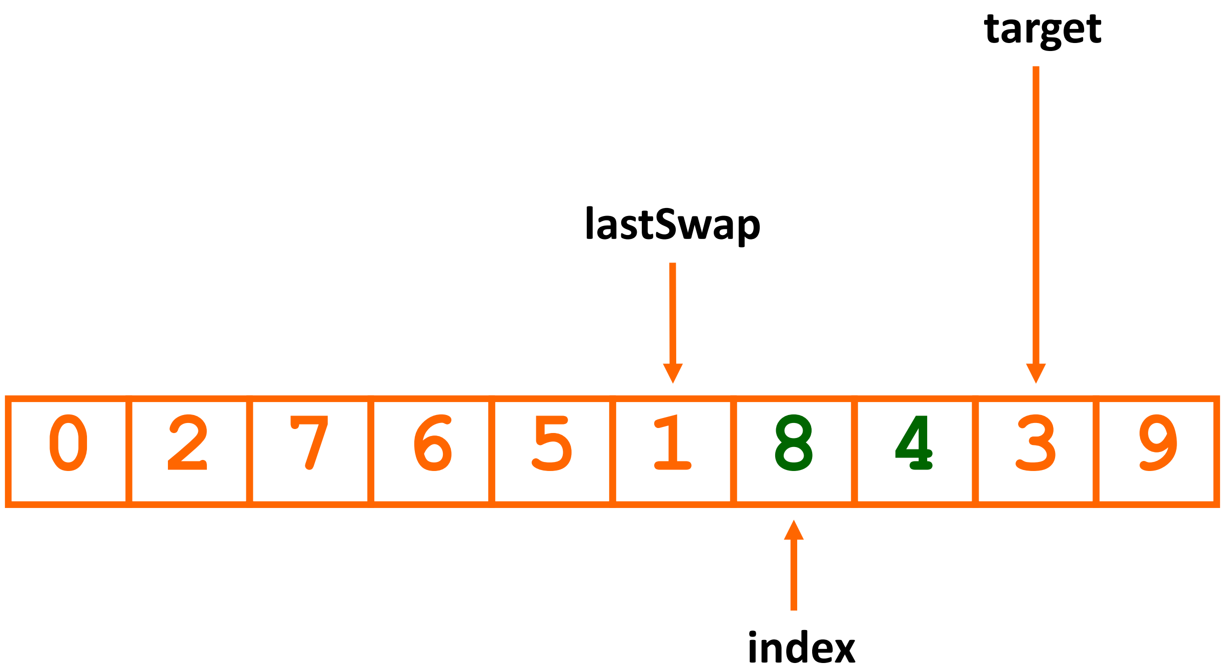
Bubble Sort Animation



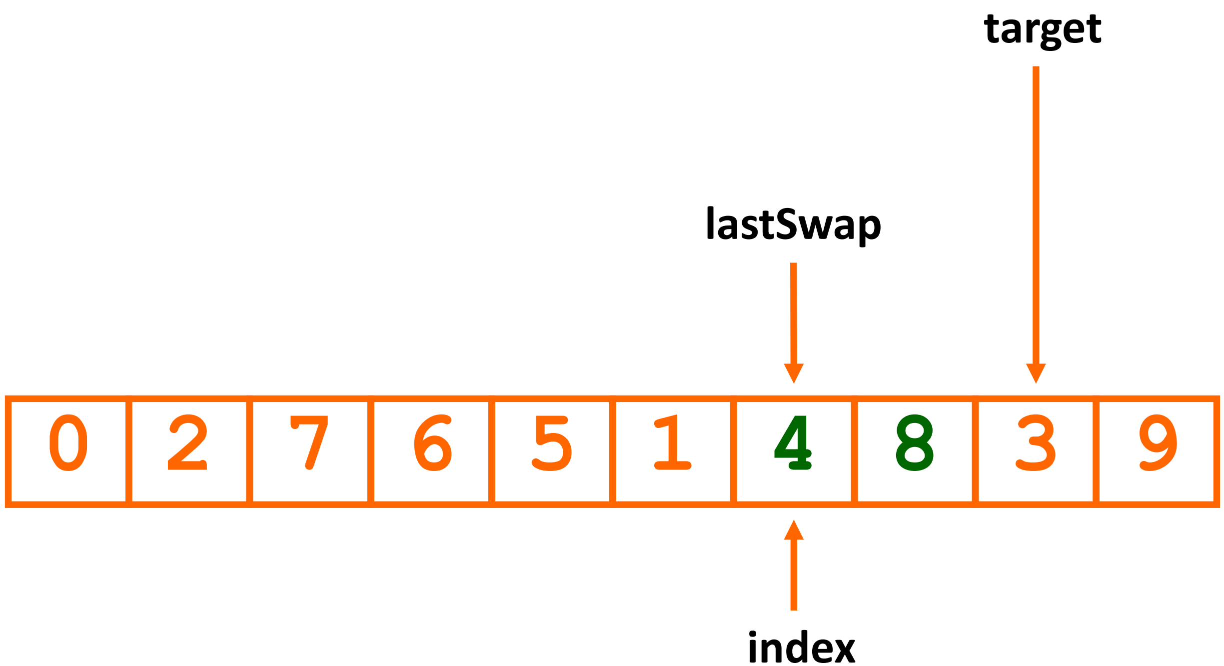
Bubble Sort Animation



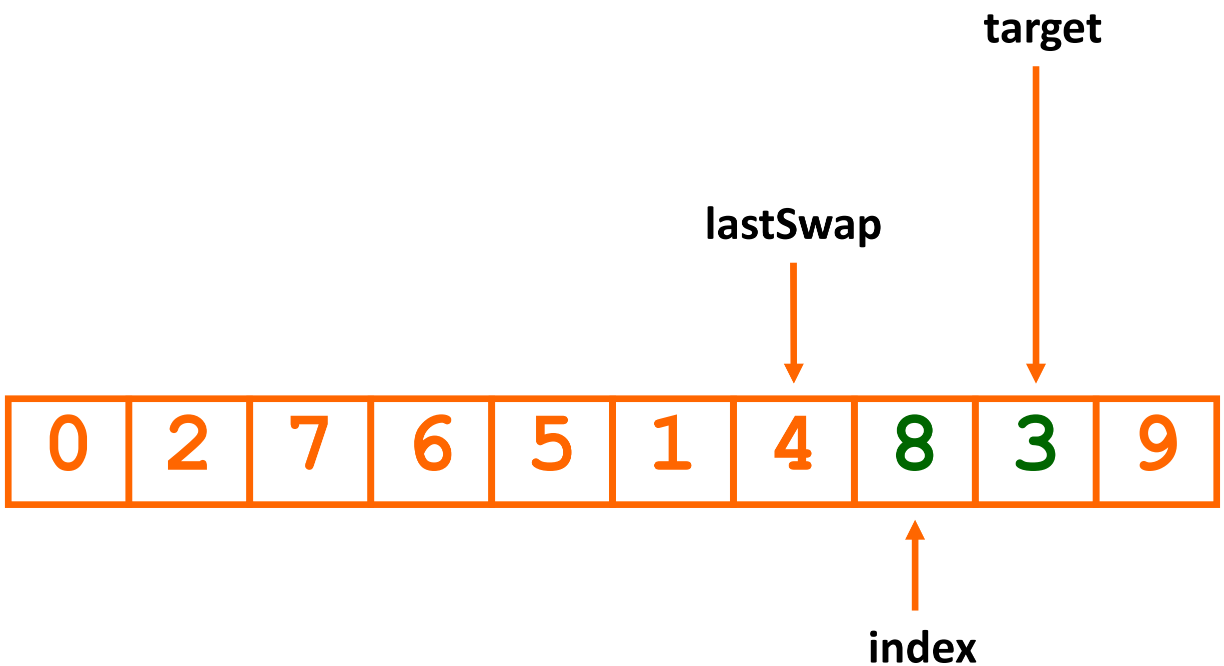
Bubble Sort Animation



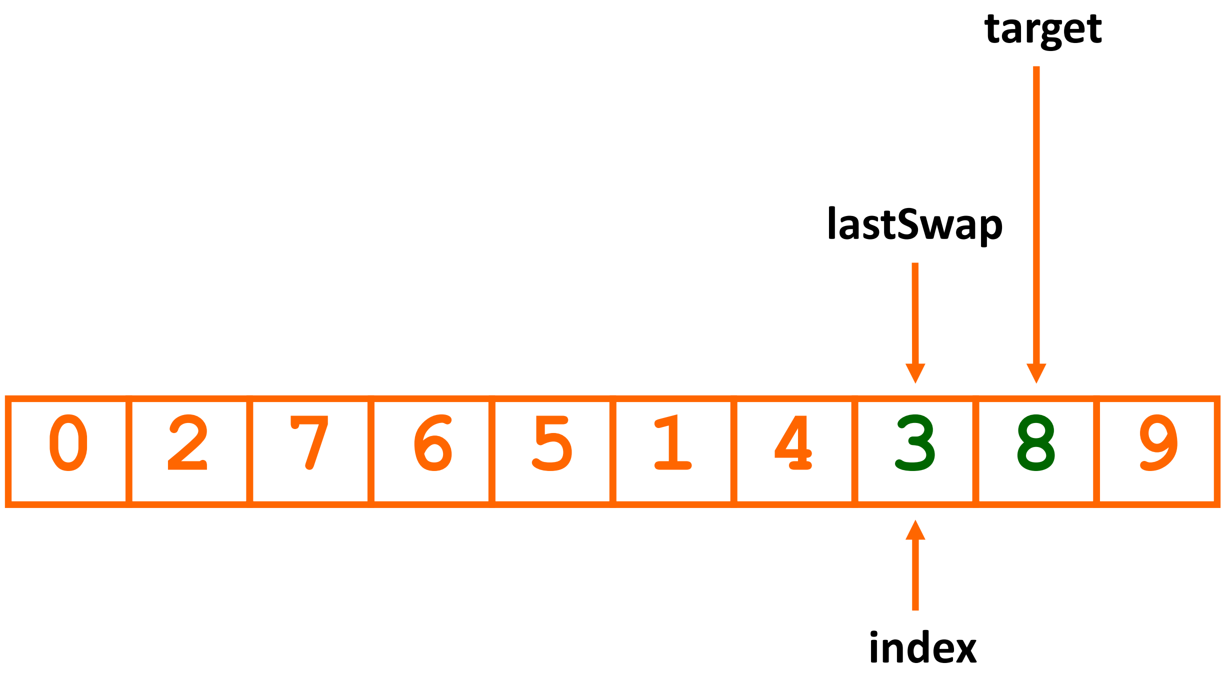
Bubble Sort Animation



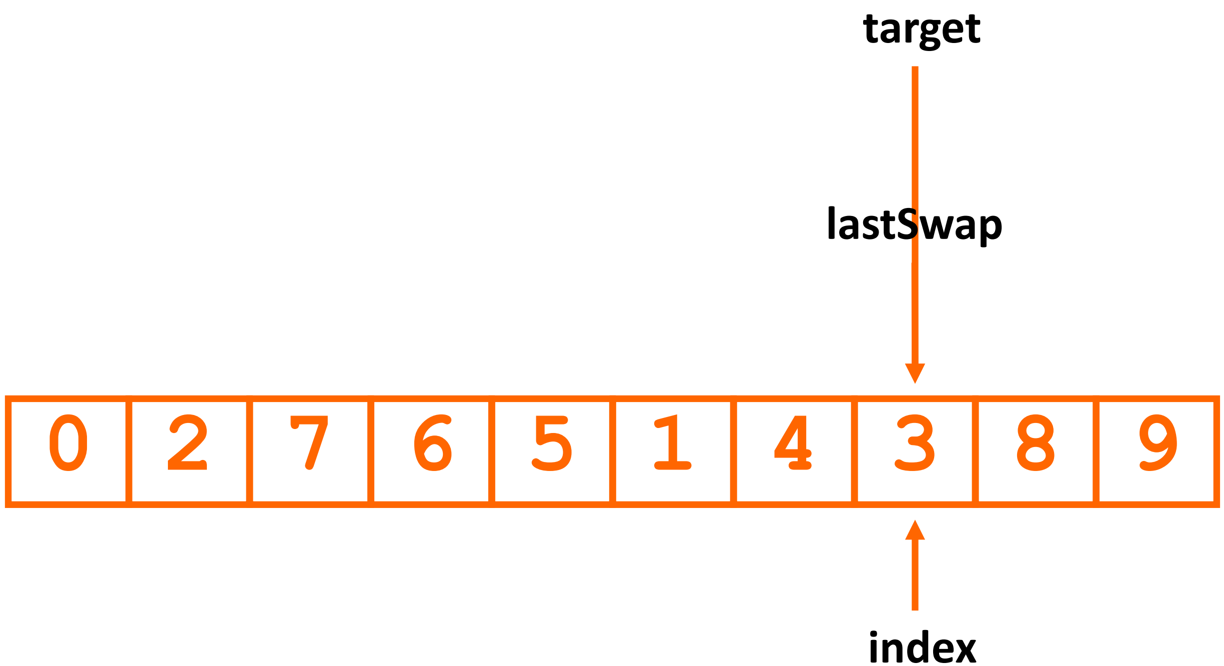
Bubble Sort Animation



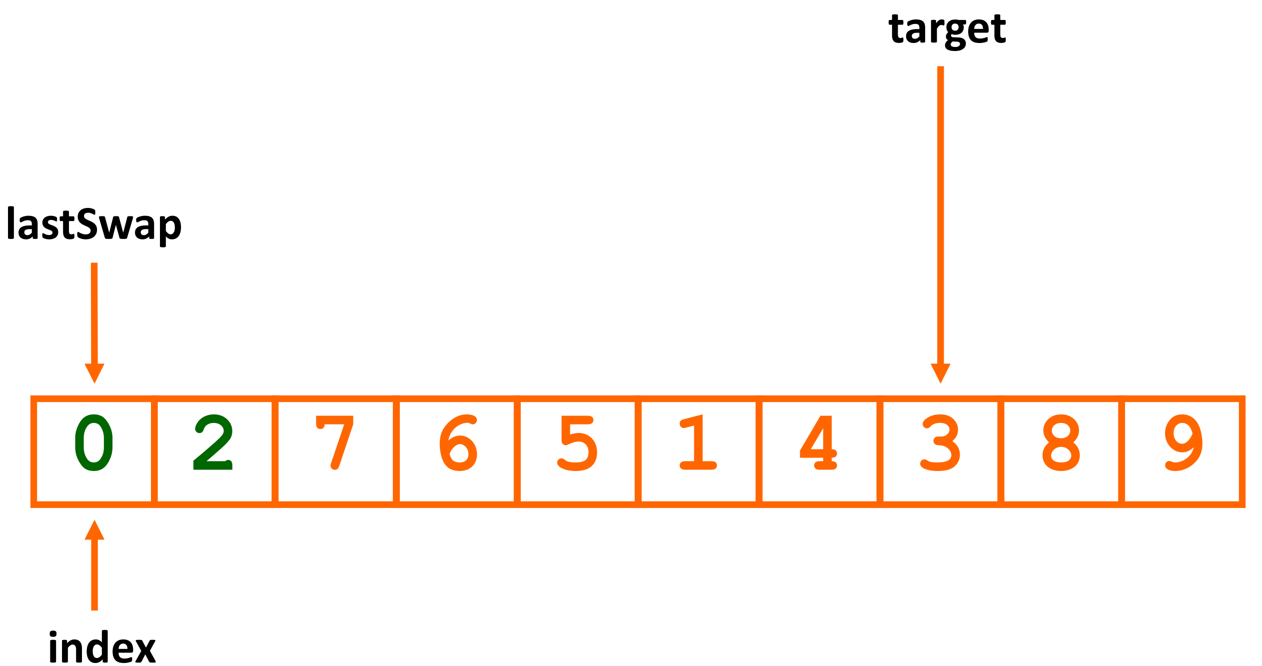
Bubble Sort Animation



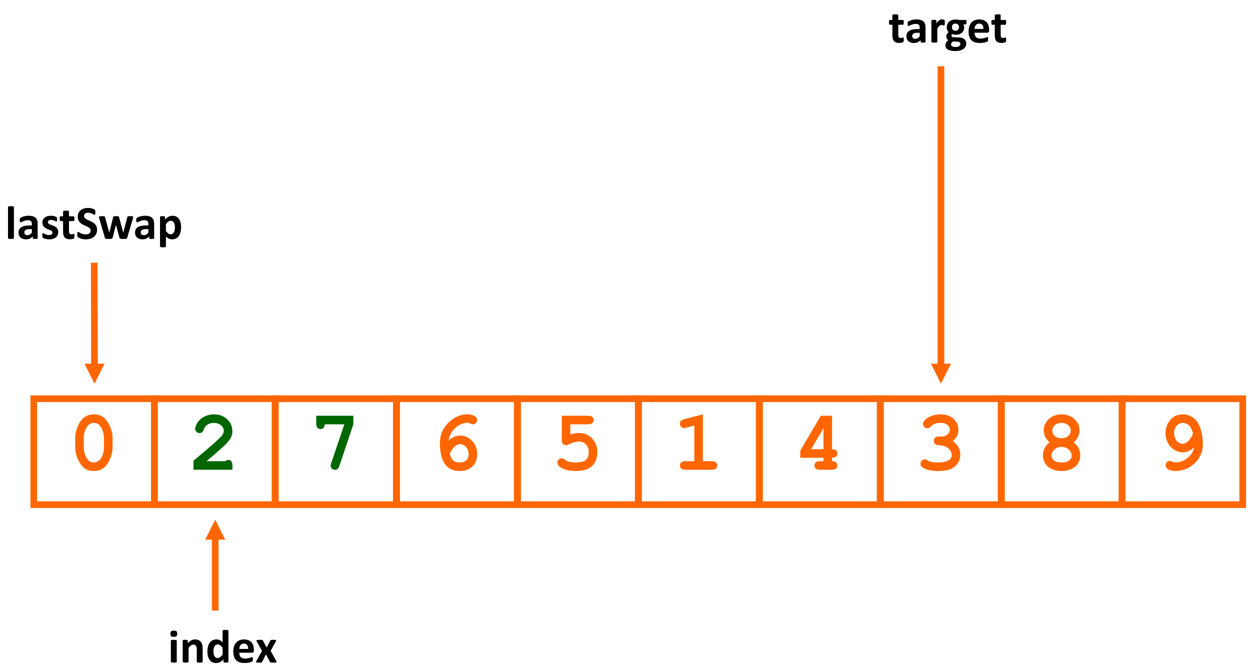
Bubble Sort Animation



Bubble Sort Animation

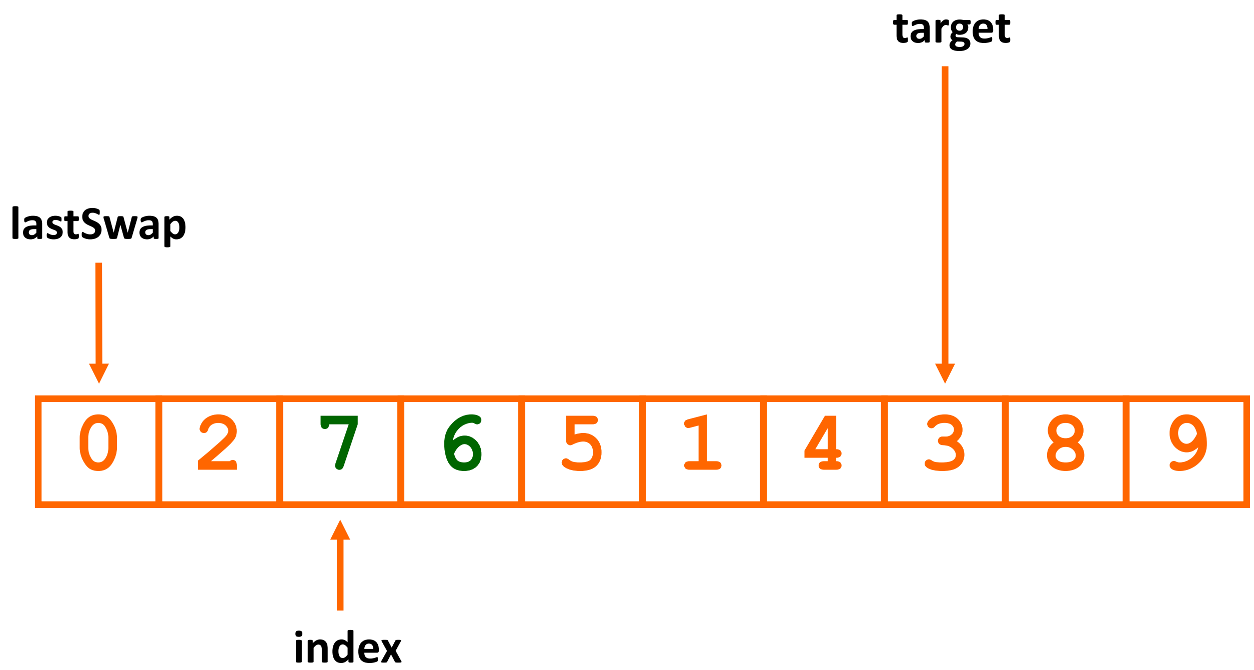


Bubble Sort Animation



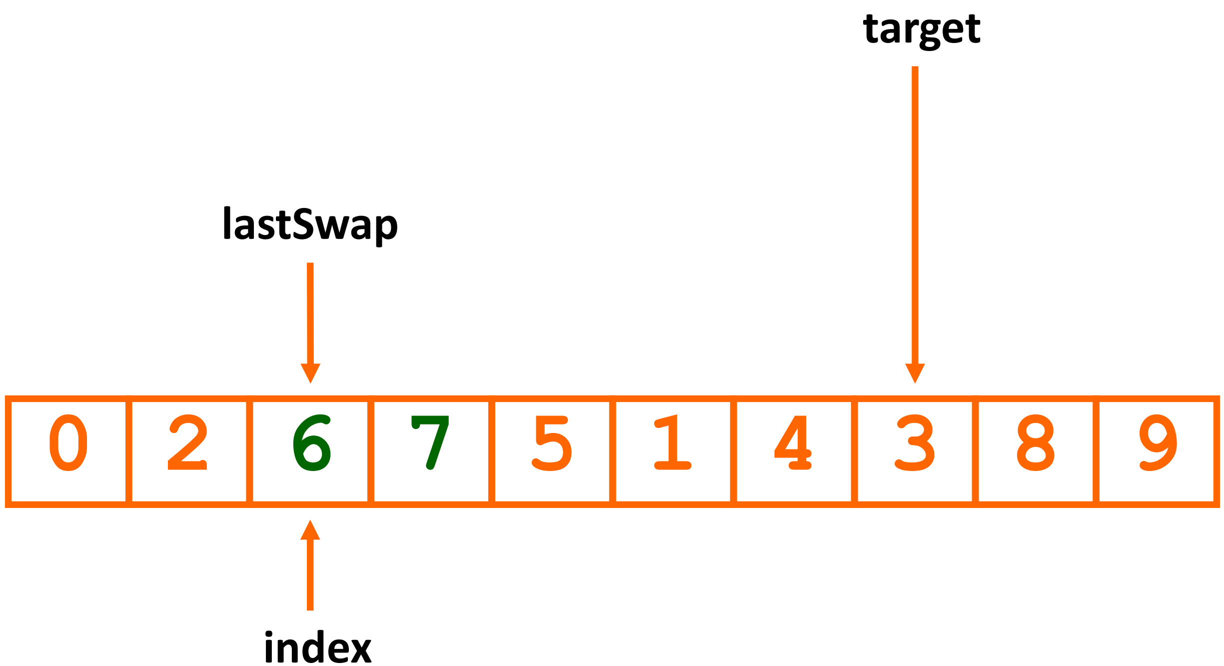
Murdoch UNIVERSITY

Bubble Sort Animation



Murdoch UNIVERSITY

Bubble Sort Animation



Bubble Sort Animation

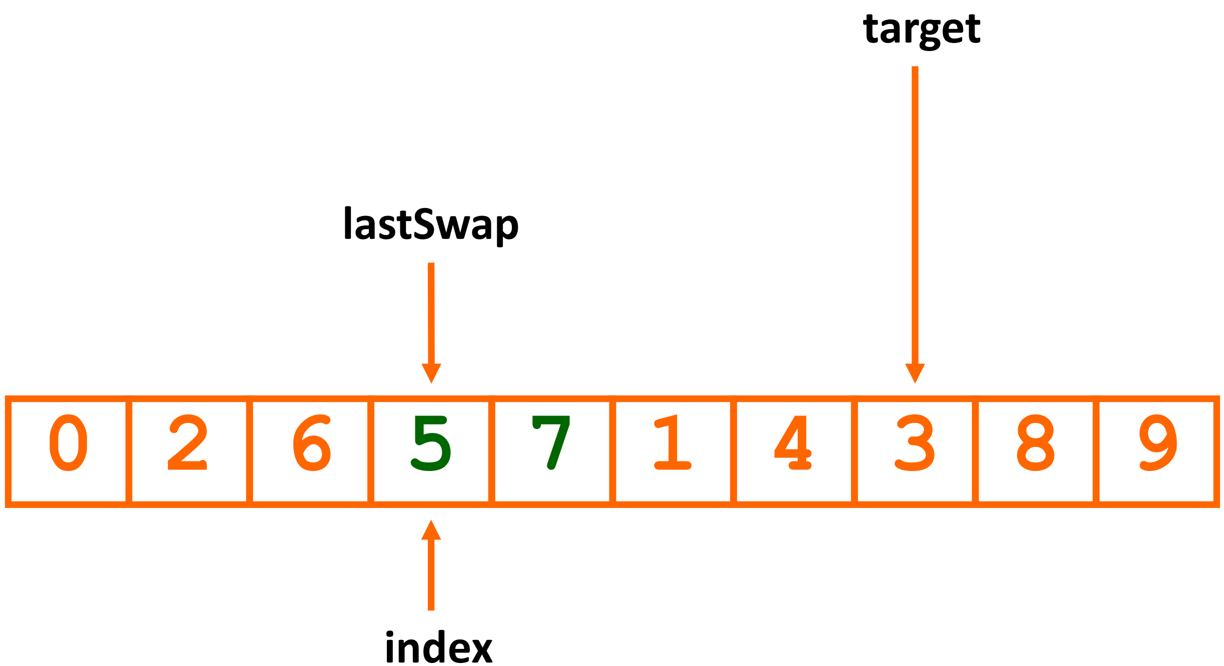
target

lastSwap

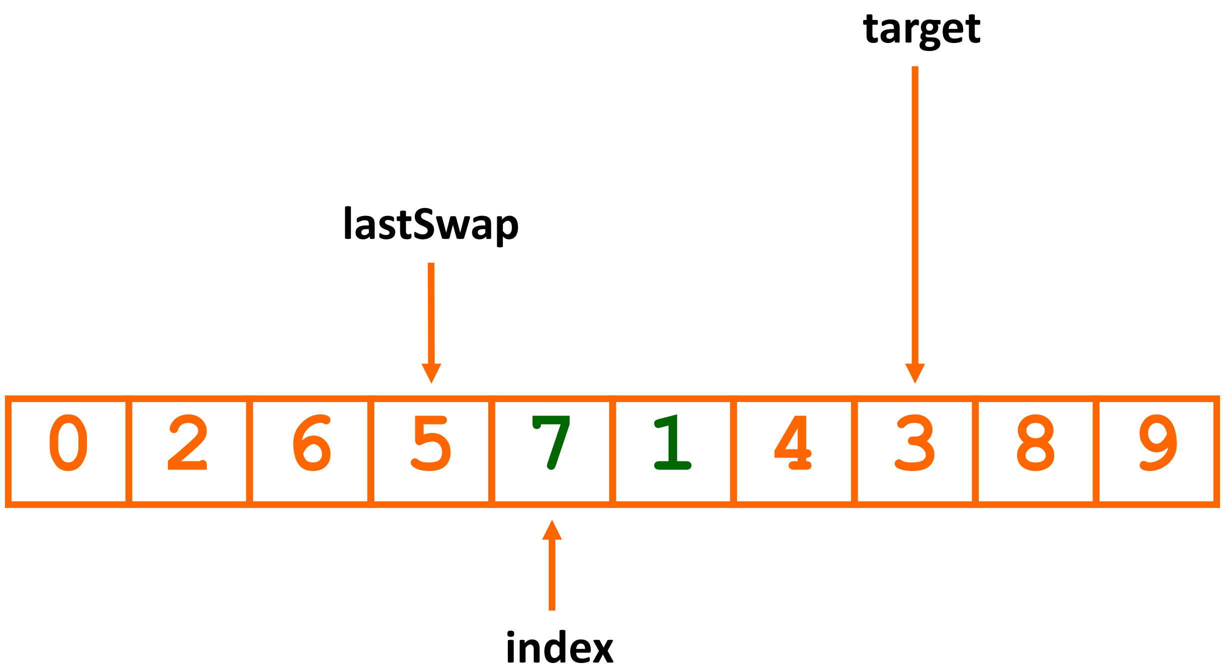
0	2	6	7	5	1	4	3	8	9

index

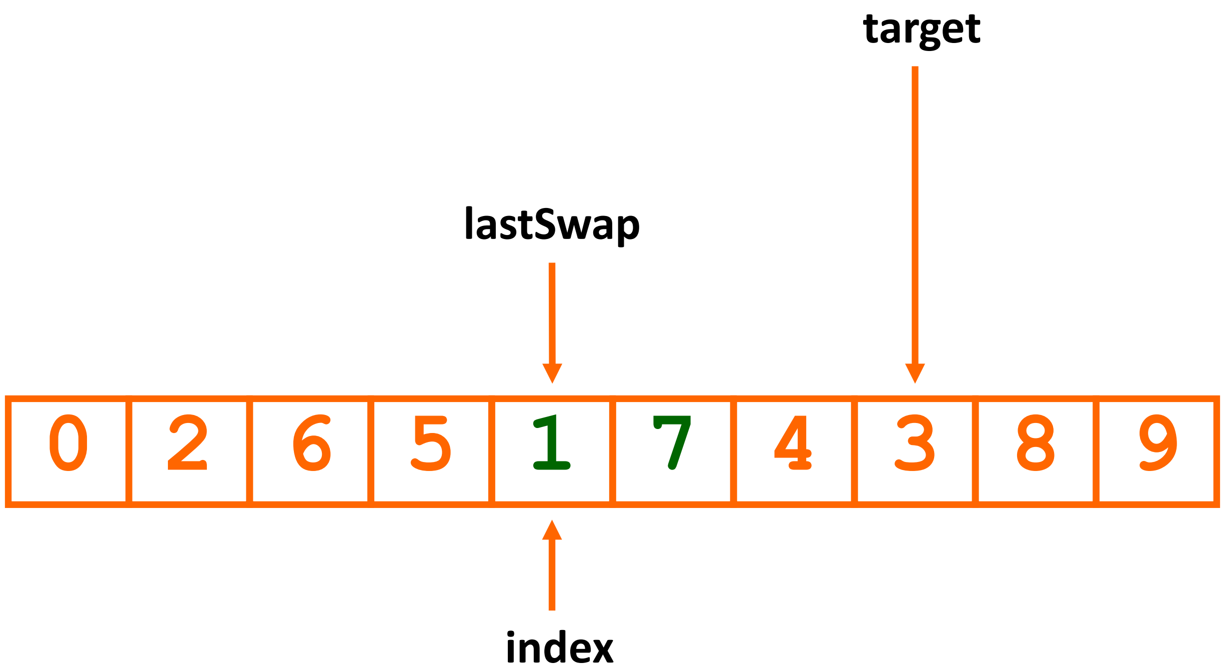
Bubble Sort Animation



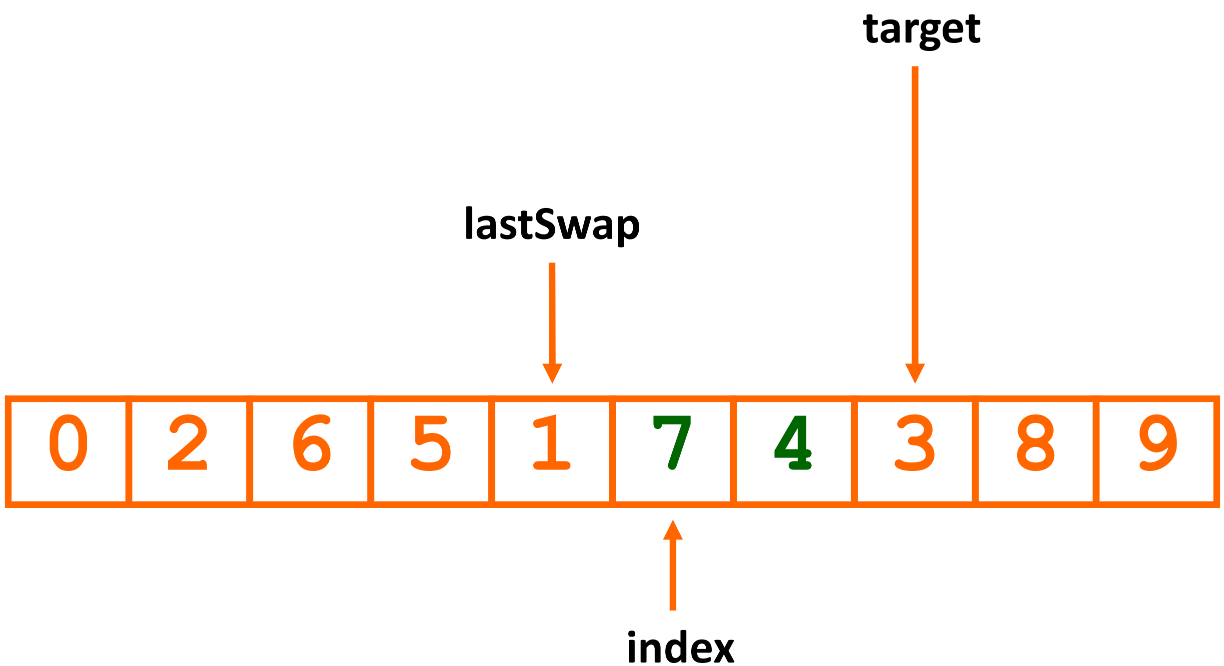
Bubble Sort Animation



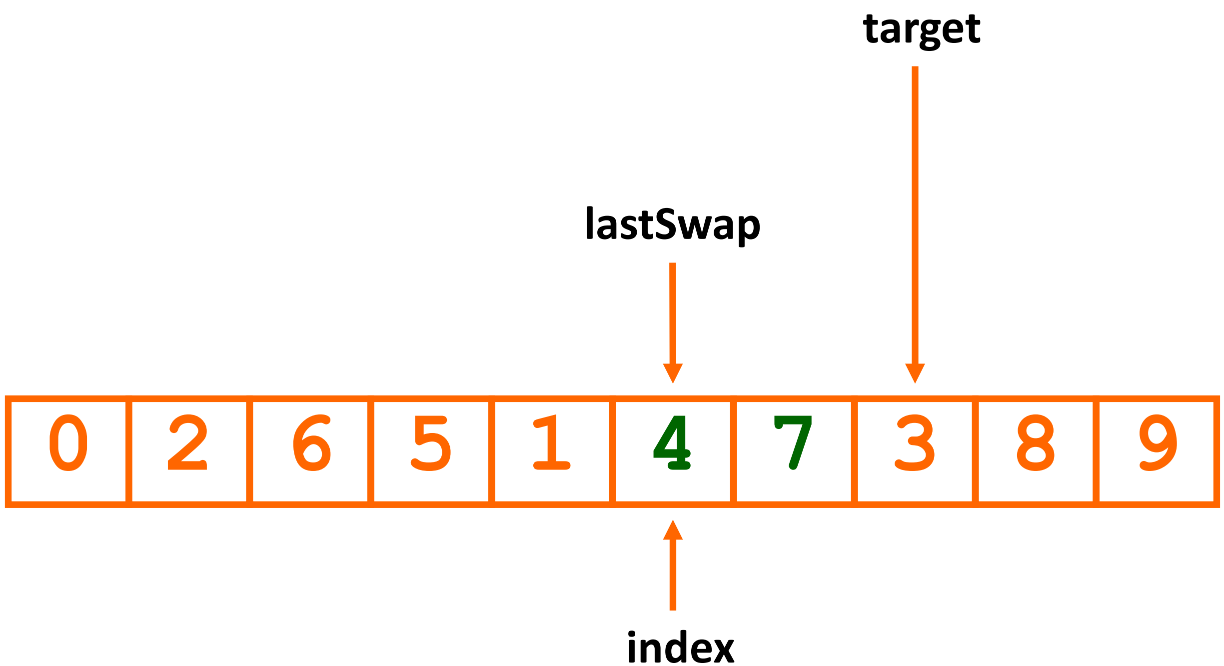
Bubble Sort Animation



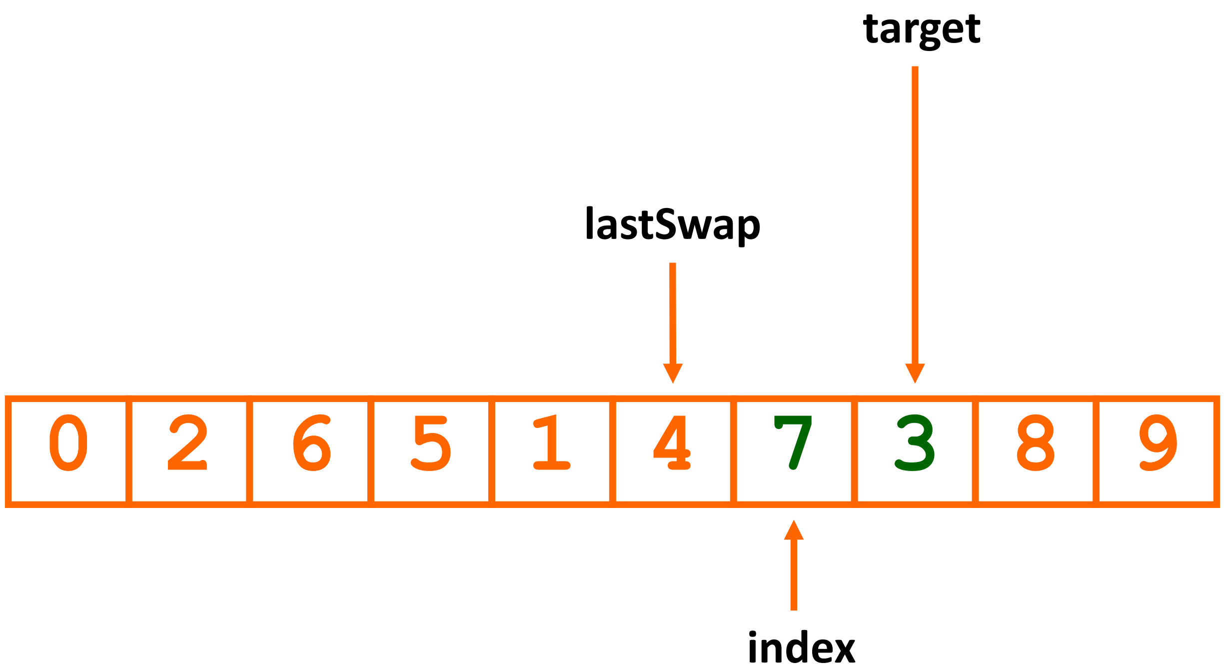
Bubble Sort Animation



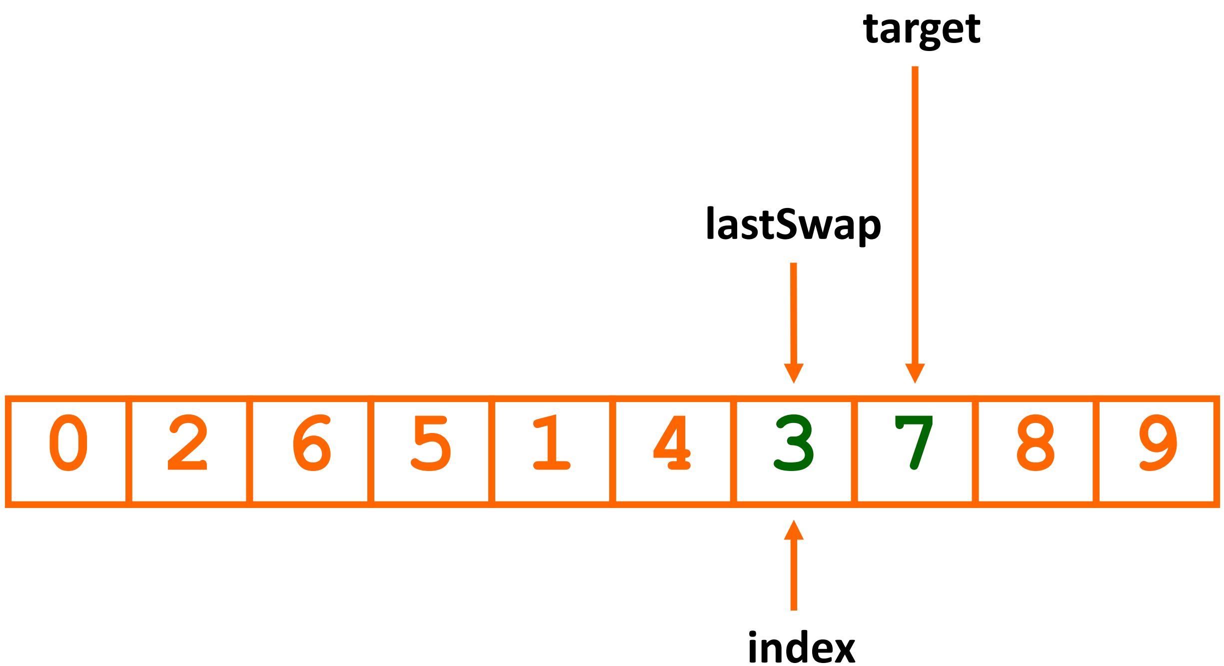
Bubble Sort Animation



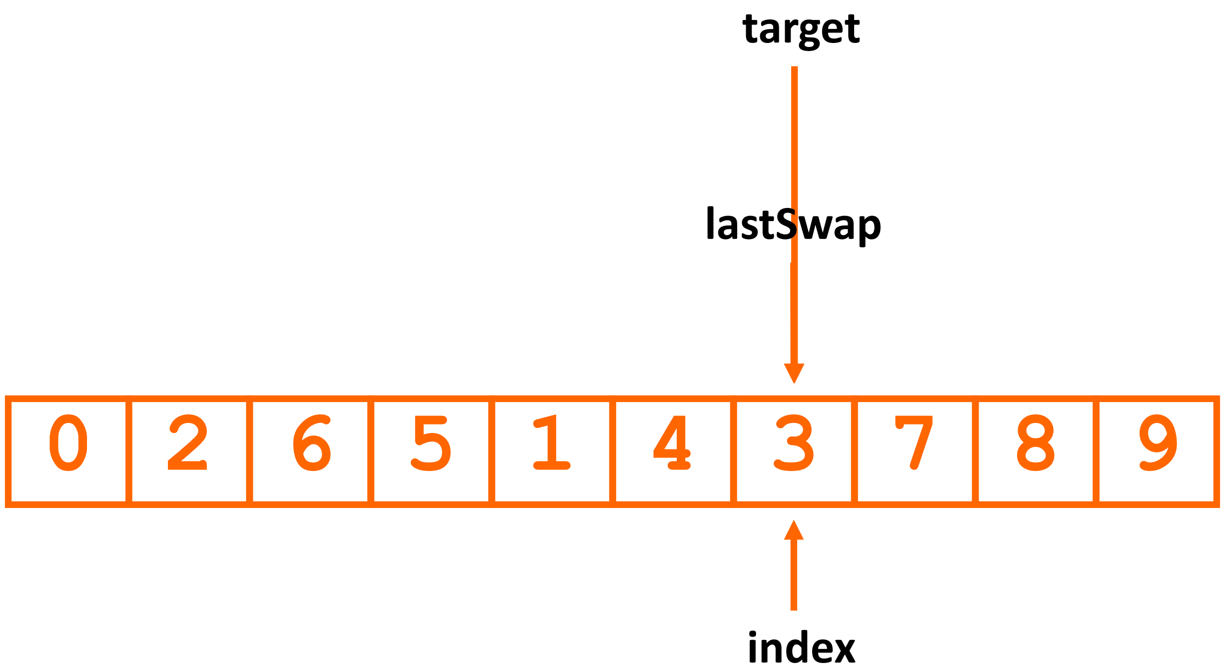
Bubble Sort Animation



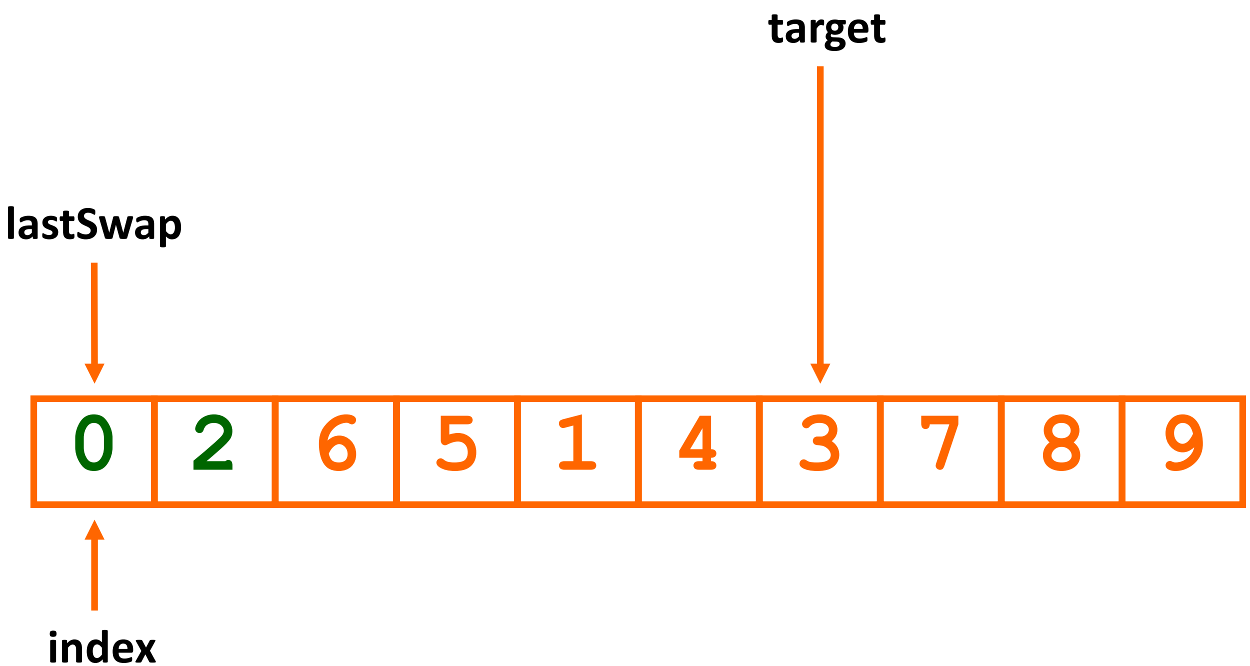
Bubble Sort Animation



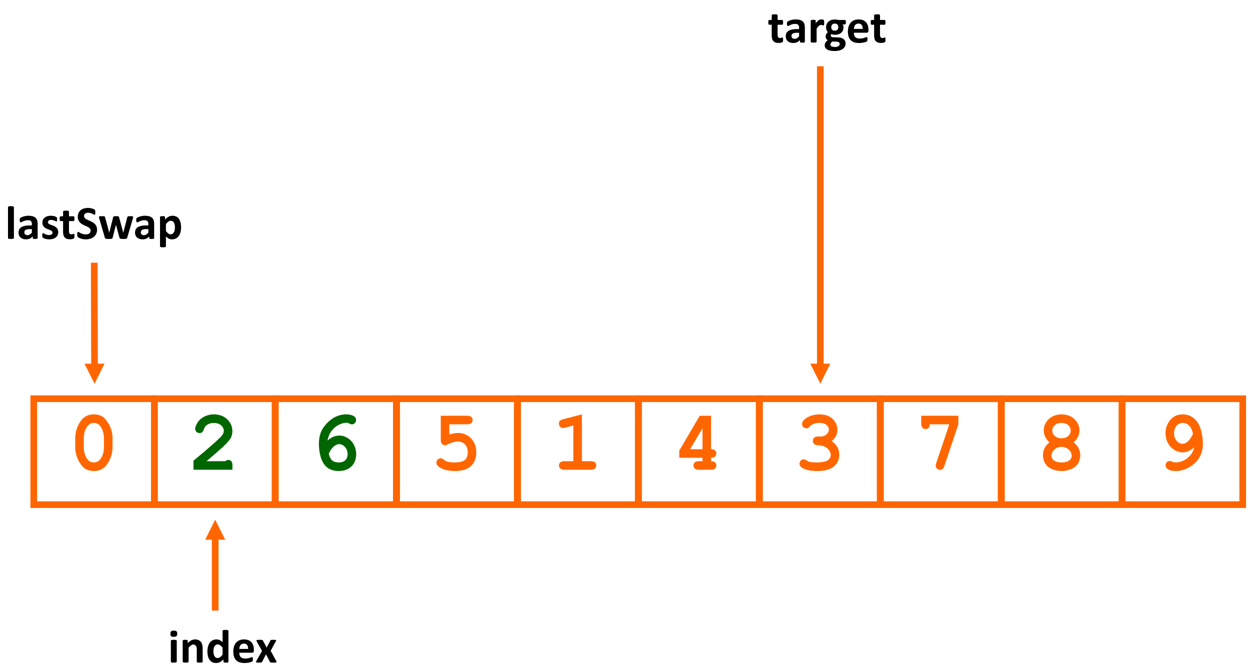
Bubble Sort Animation



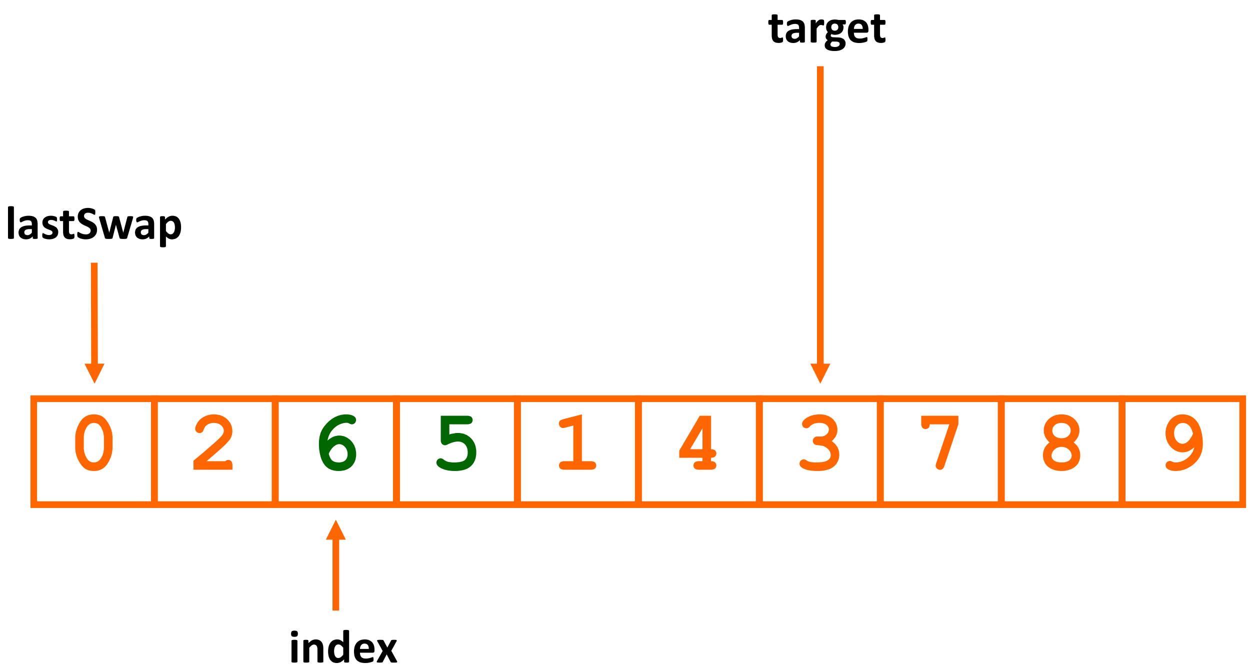
Bubble Sort Animation



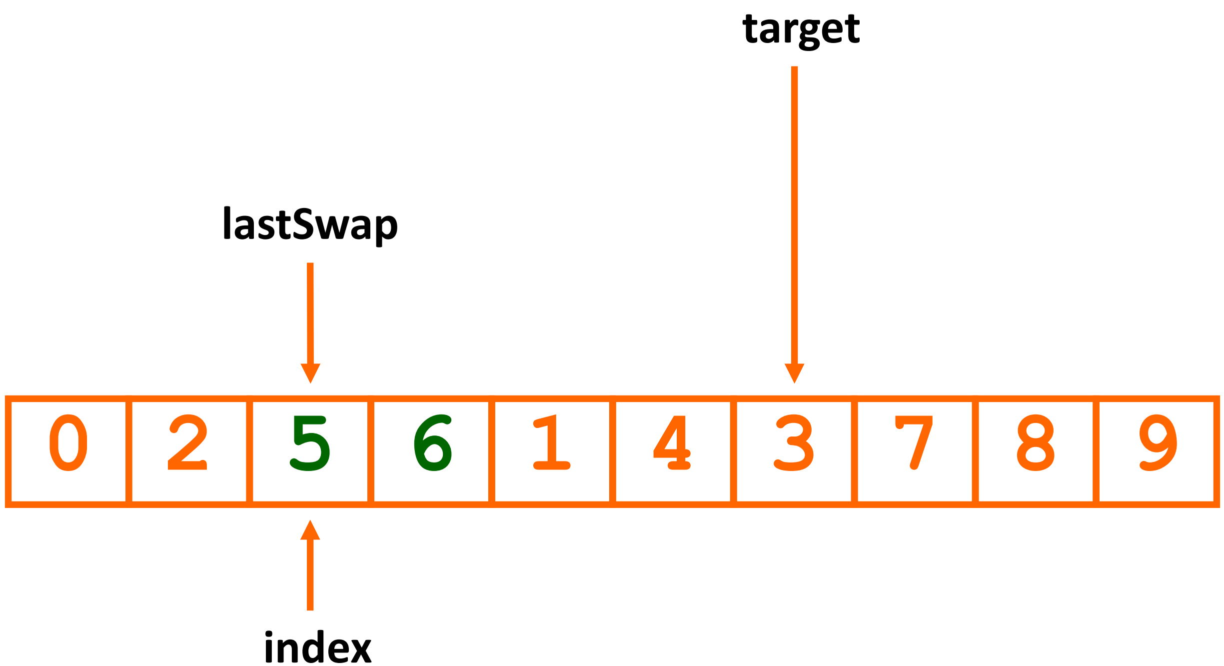
Bubble Sort Animation



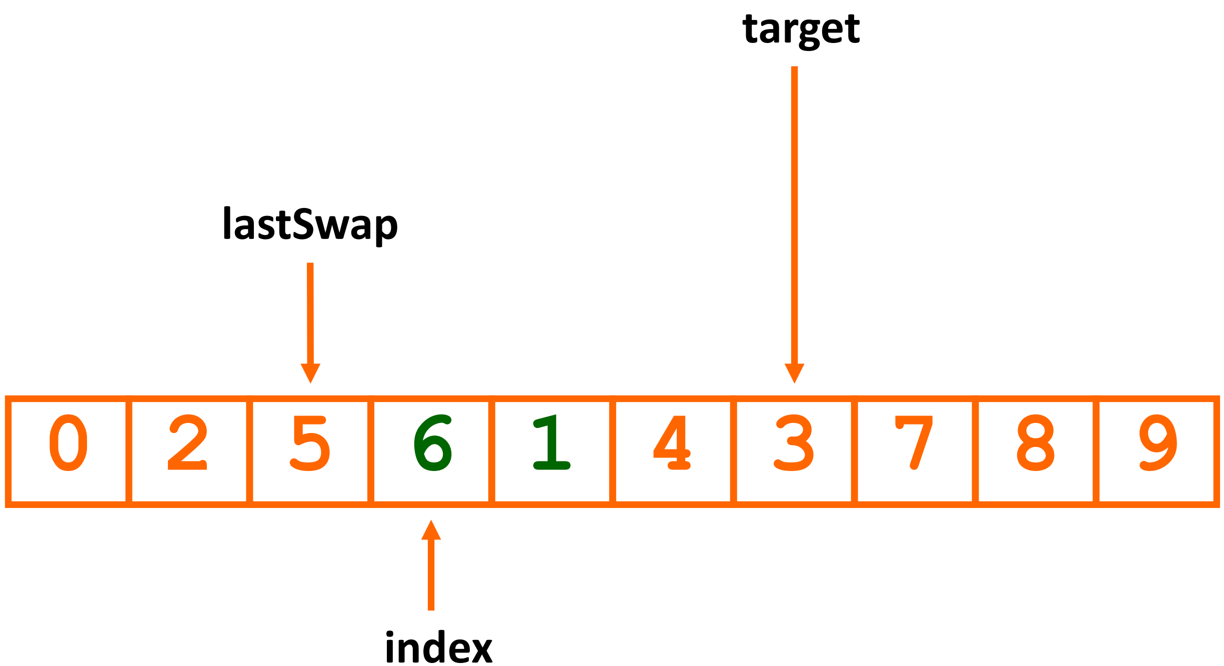
Bubble Sort Animation



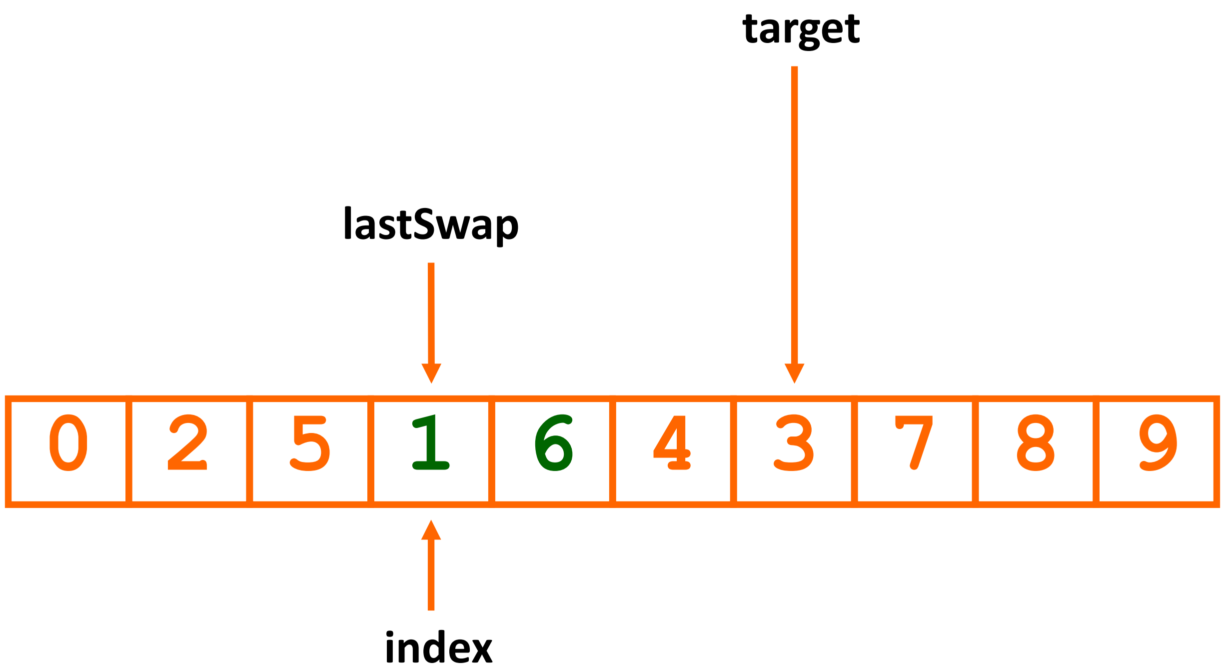
Bubble Sort Animation



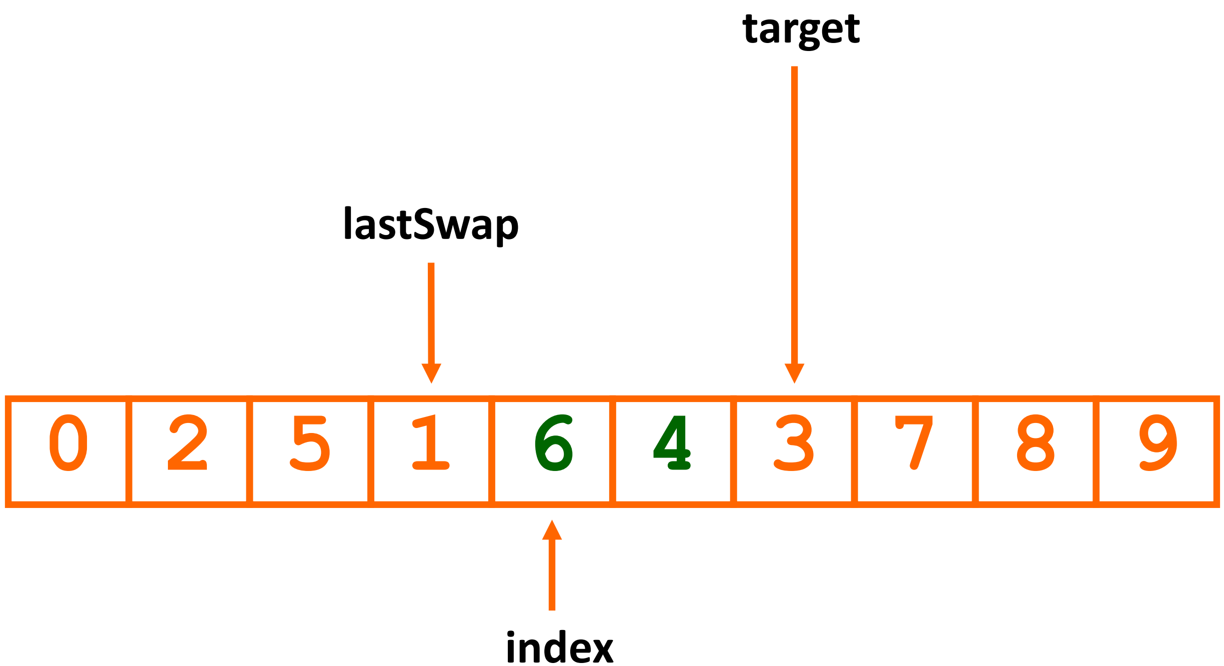
Bubble Sort Animation



Bubble Sort Animation

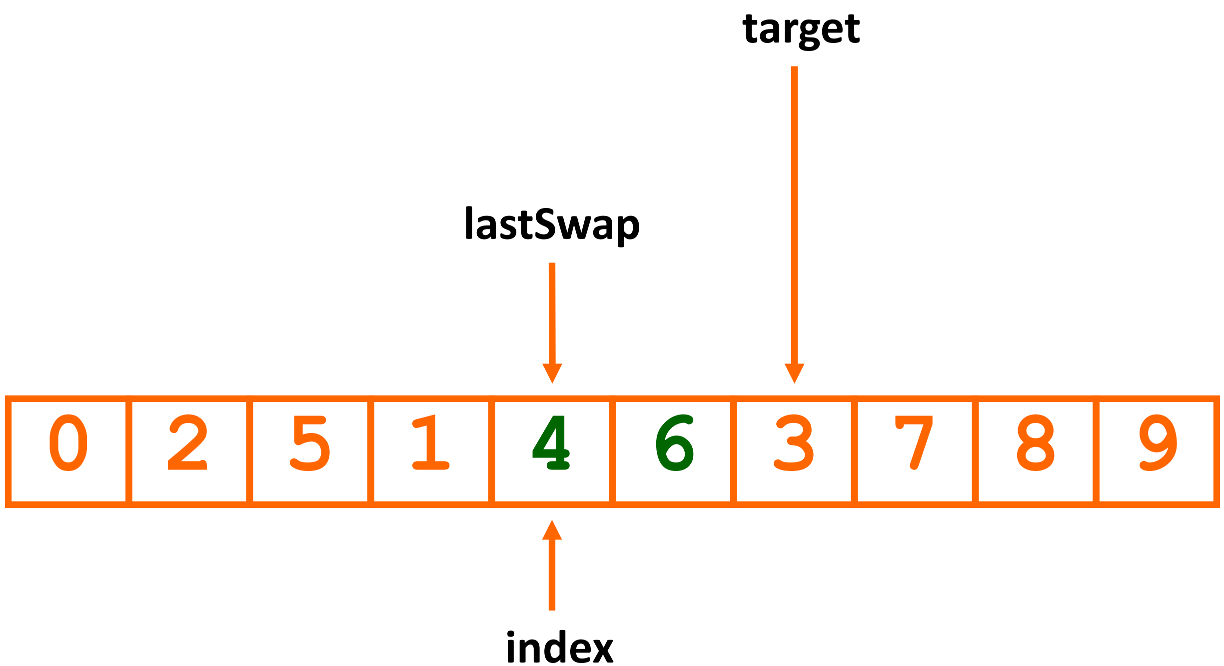


Bubble Sort Animation

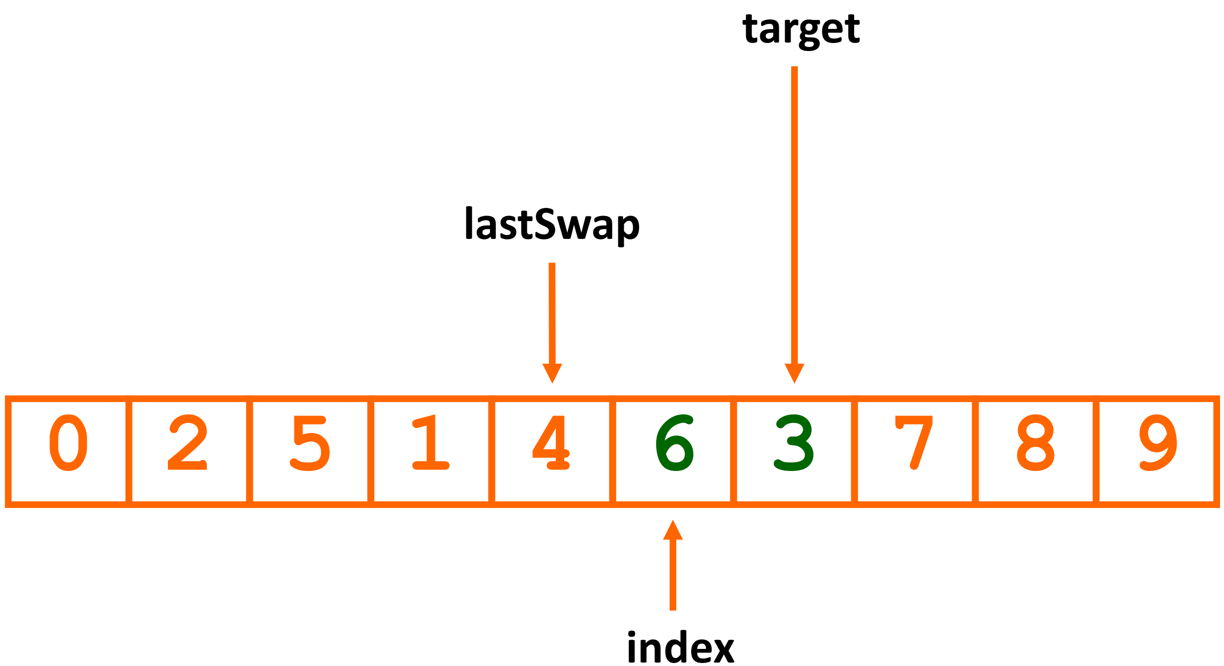


Murdoch UNIVERSITY

Bubble Sort Animation

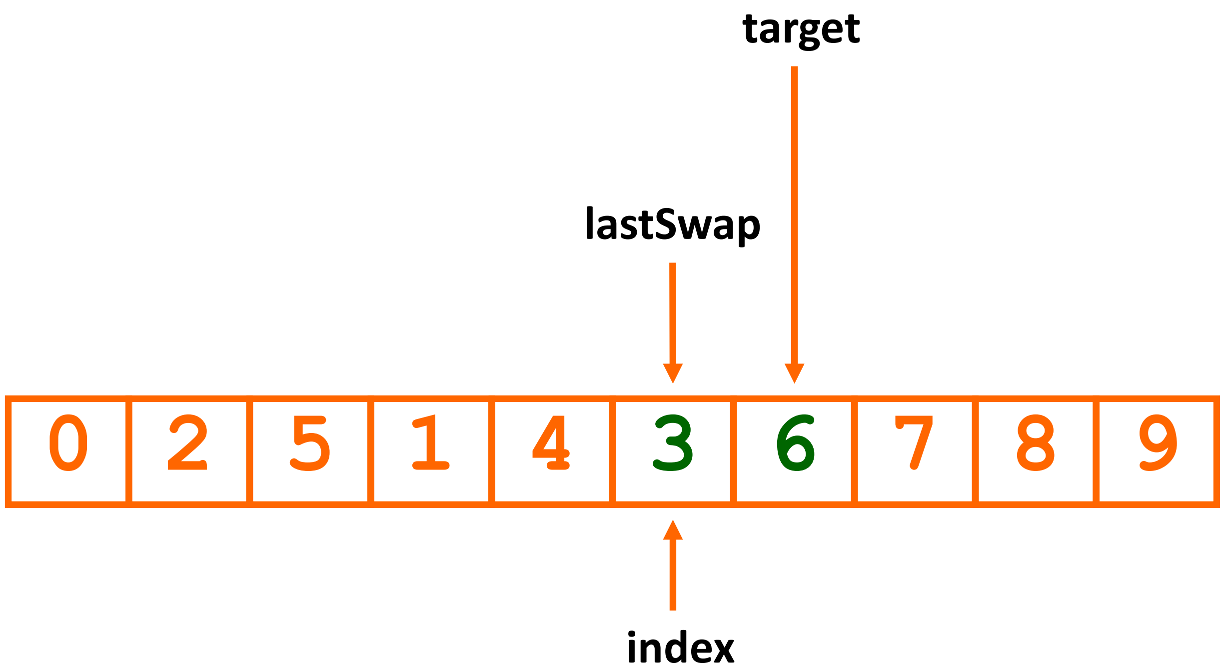


Bubble Sort Animation

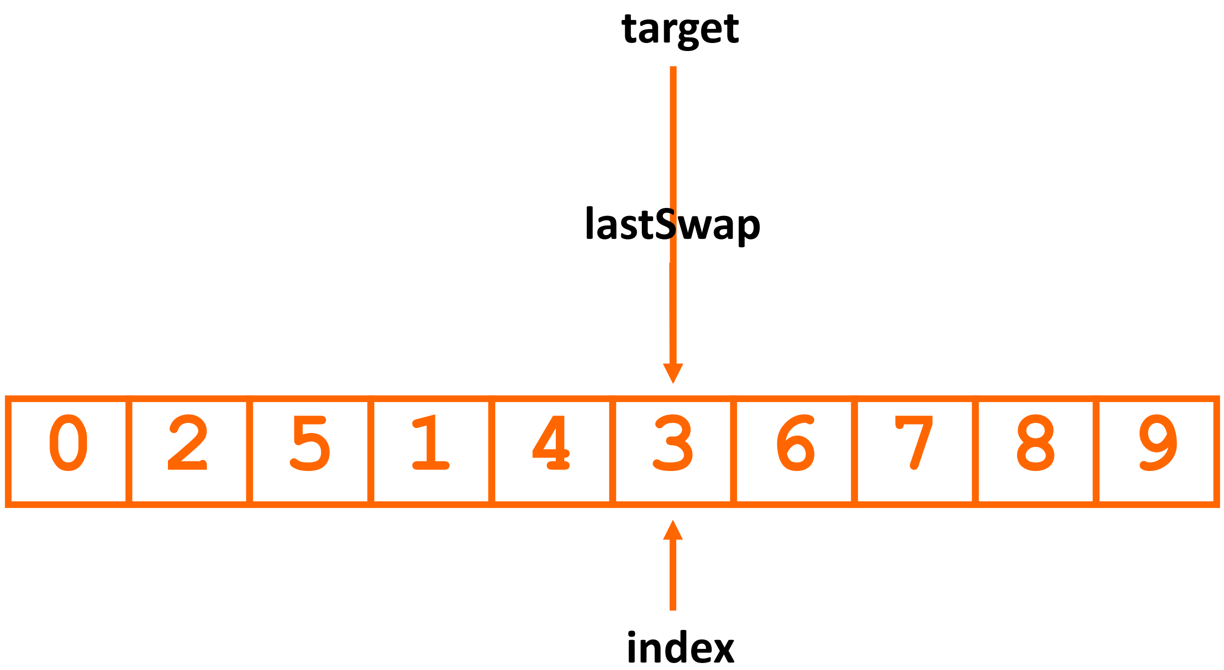


Murdoch UNIVERSITY

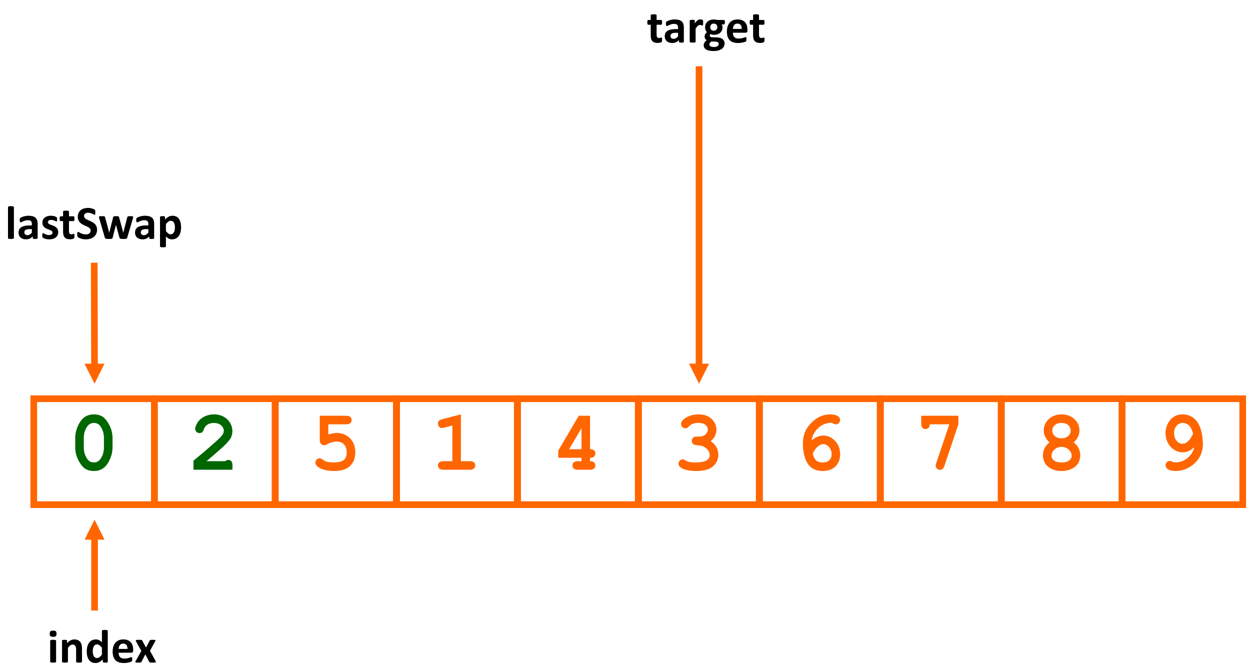
Bubble Sort Animation



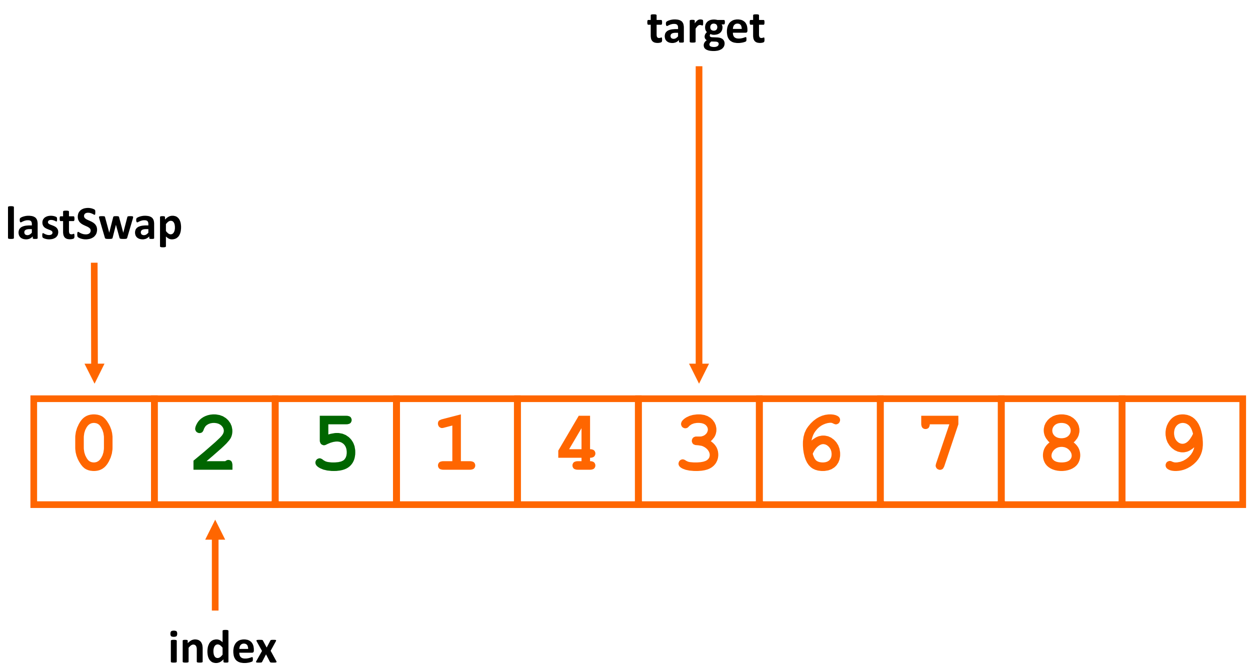
Bubble Sort Animation



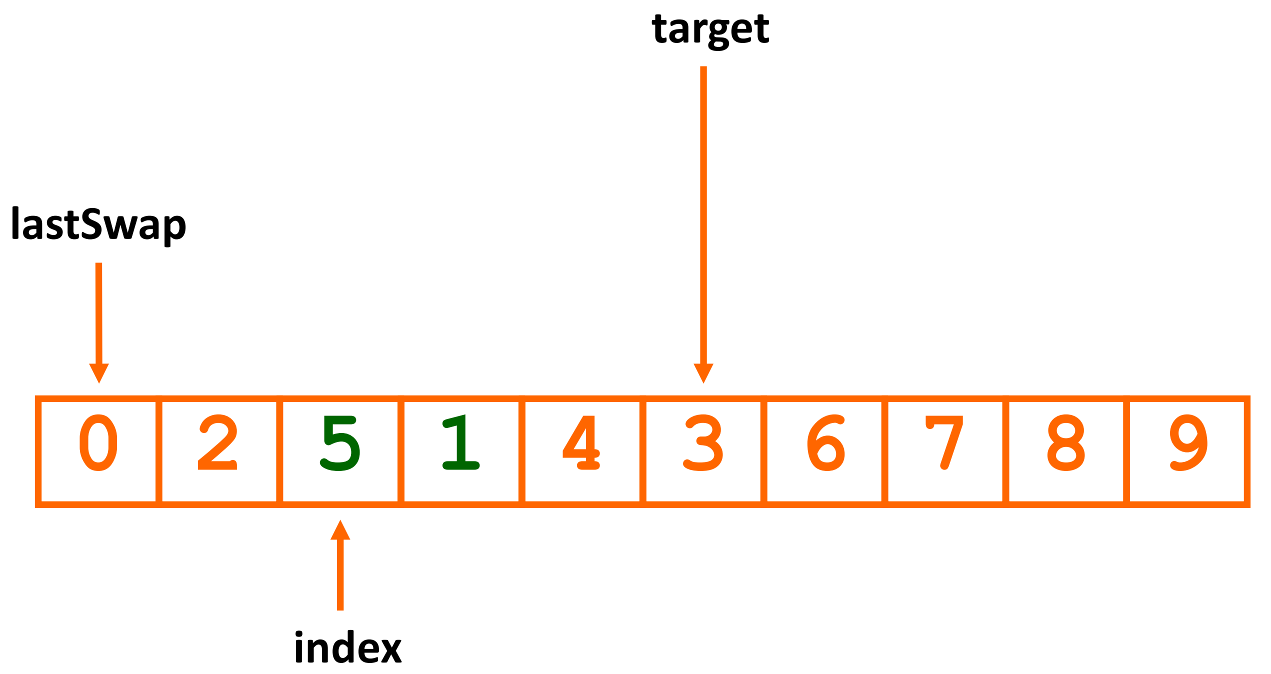
Bubble Sort Animation



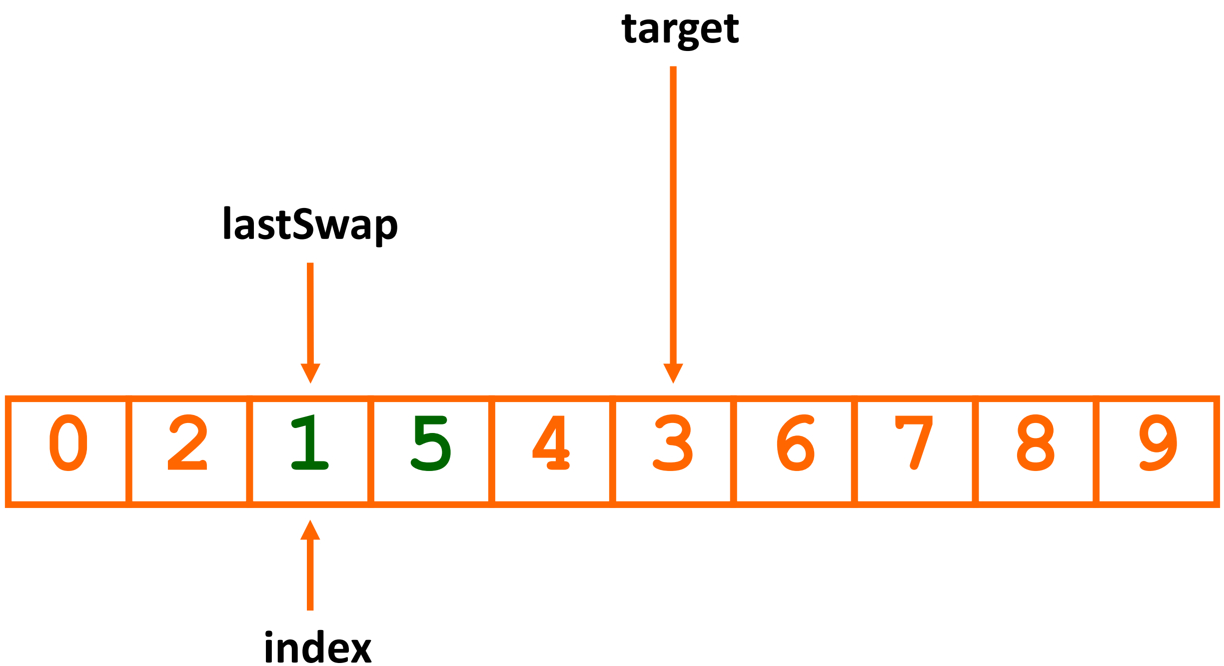
Bubble Sort Animation



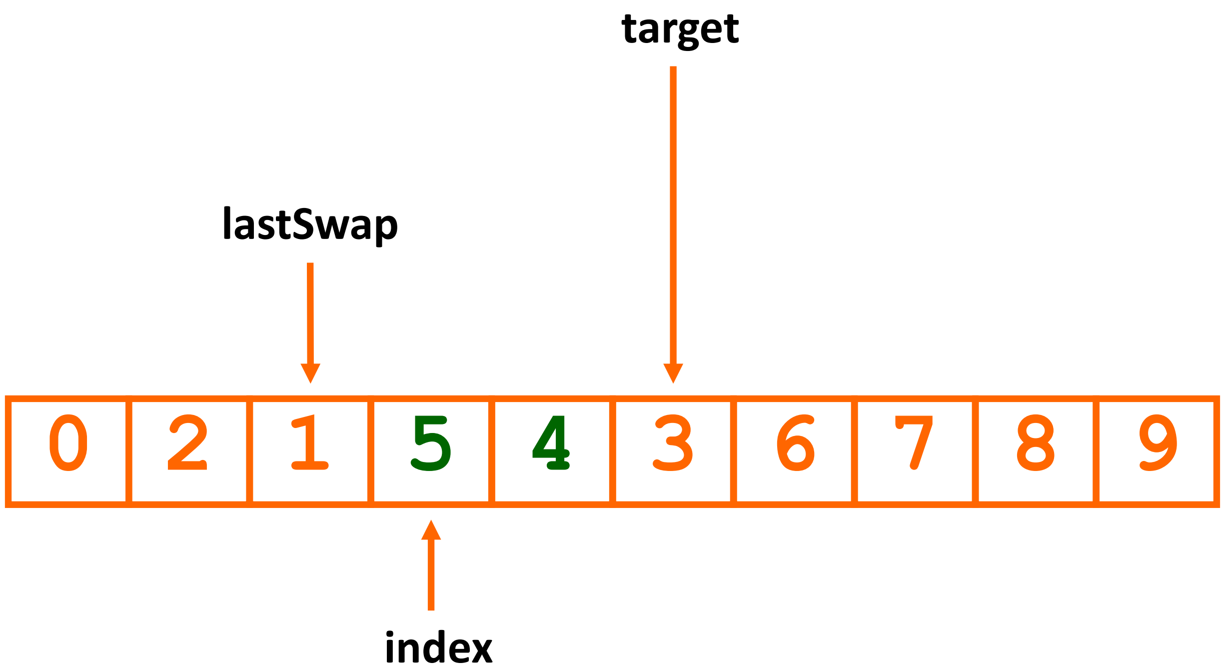
Bubble Sort Animation



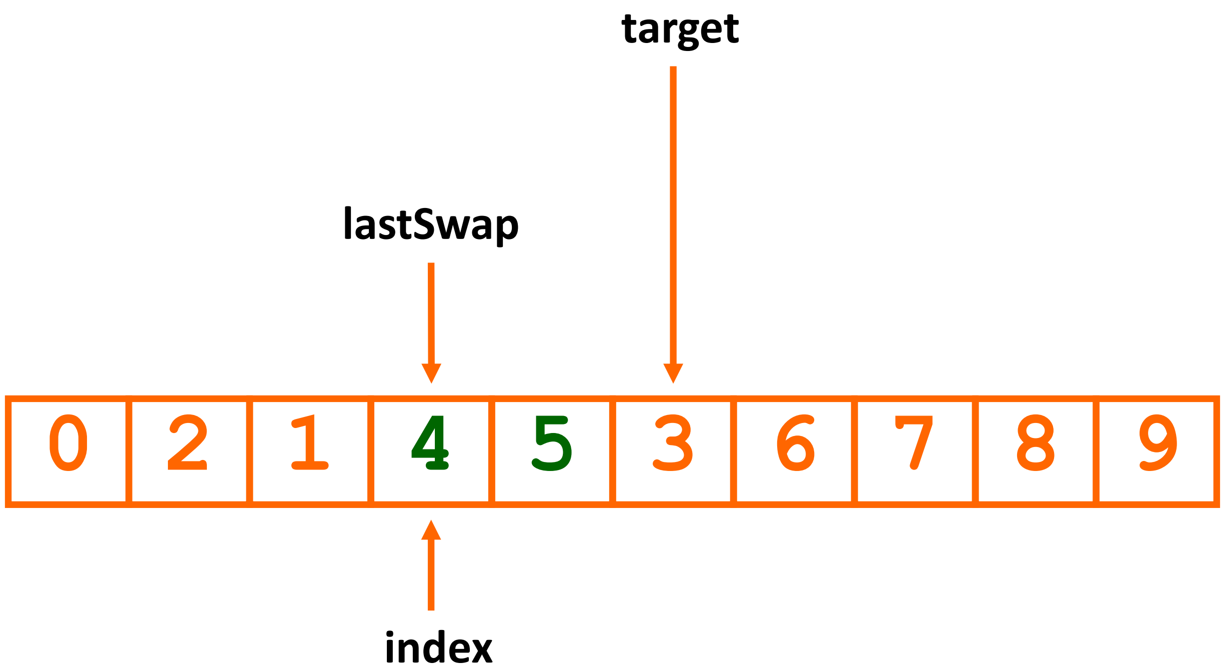
Bubble Sort Animation



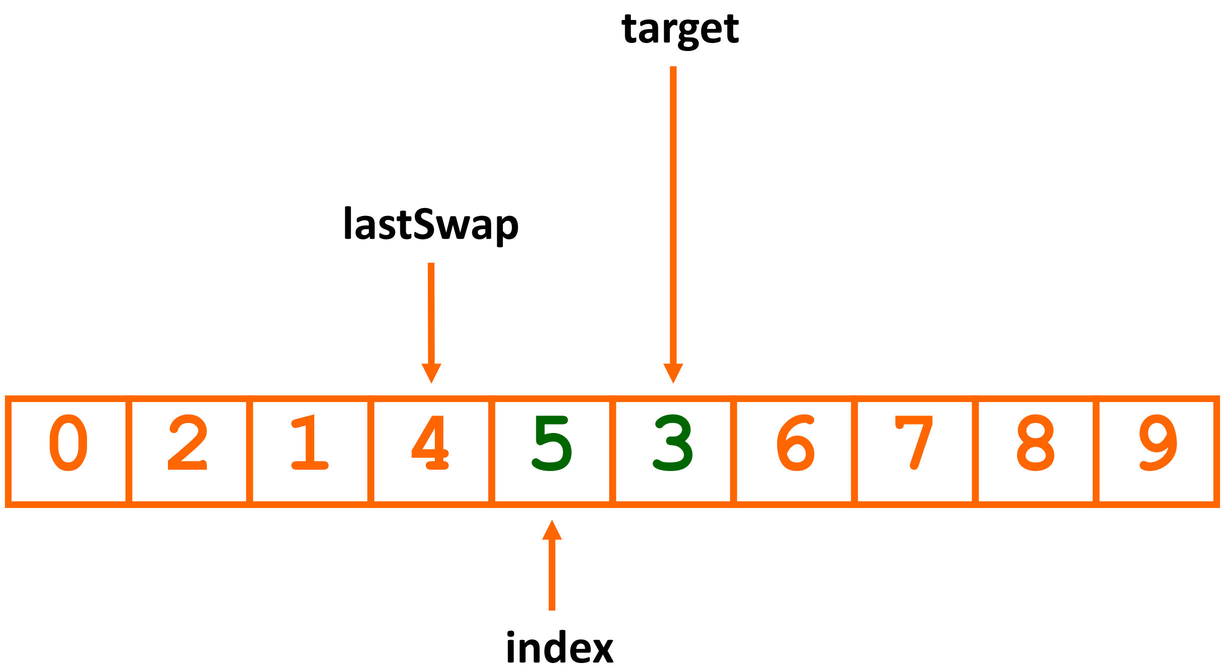
Bubble Sort Animation



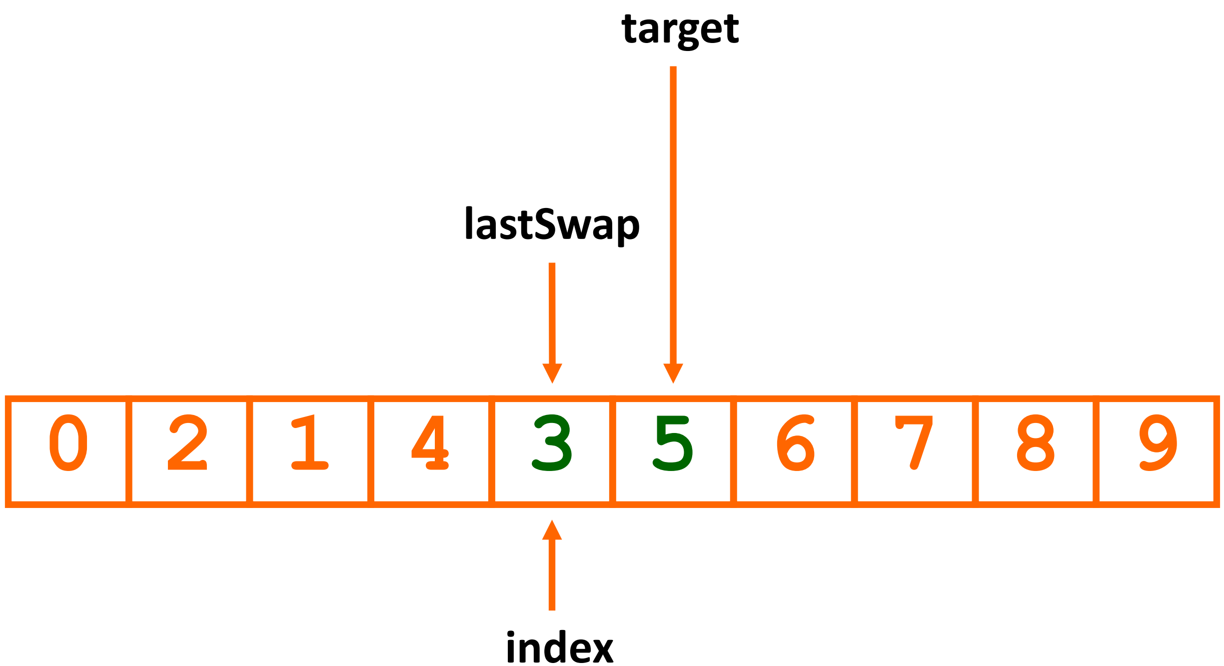
Bubble Sort Animation



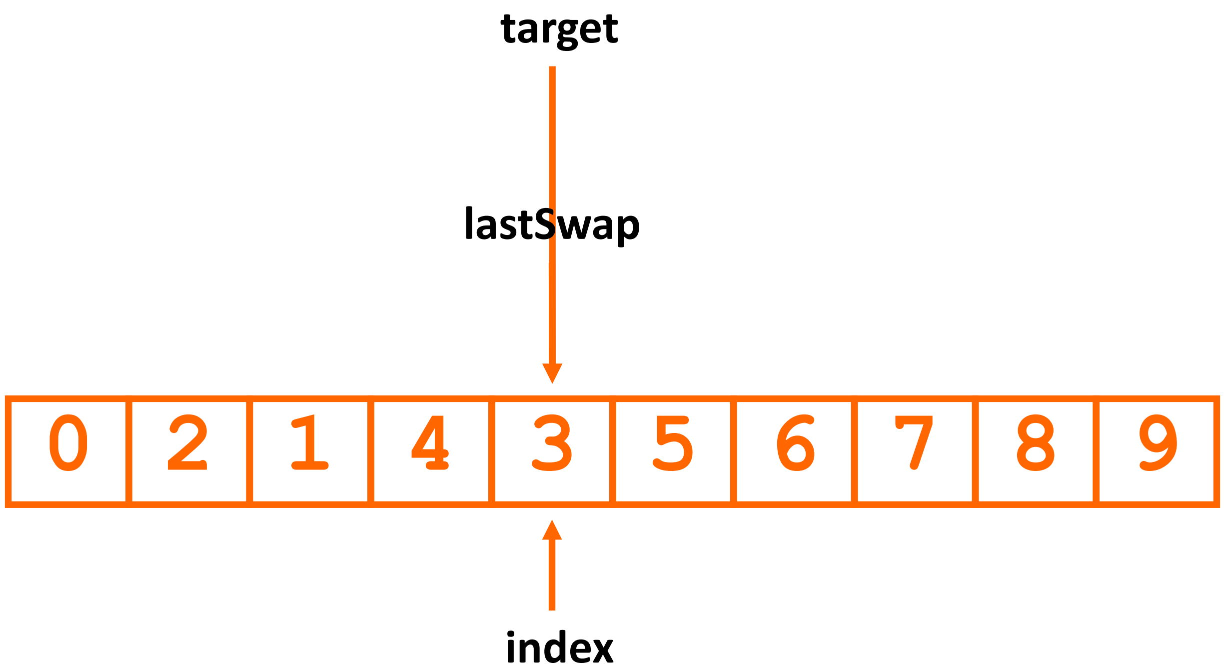
Bubble Sort Animation



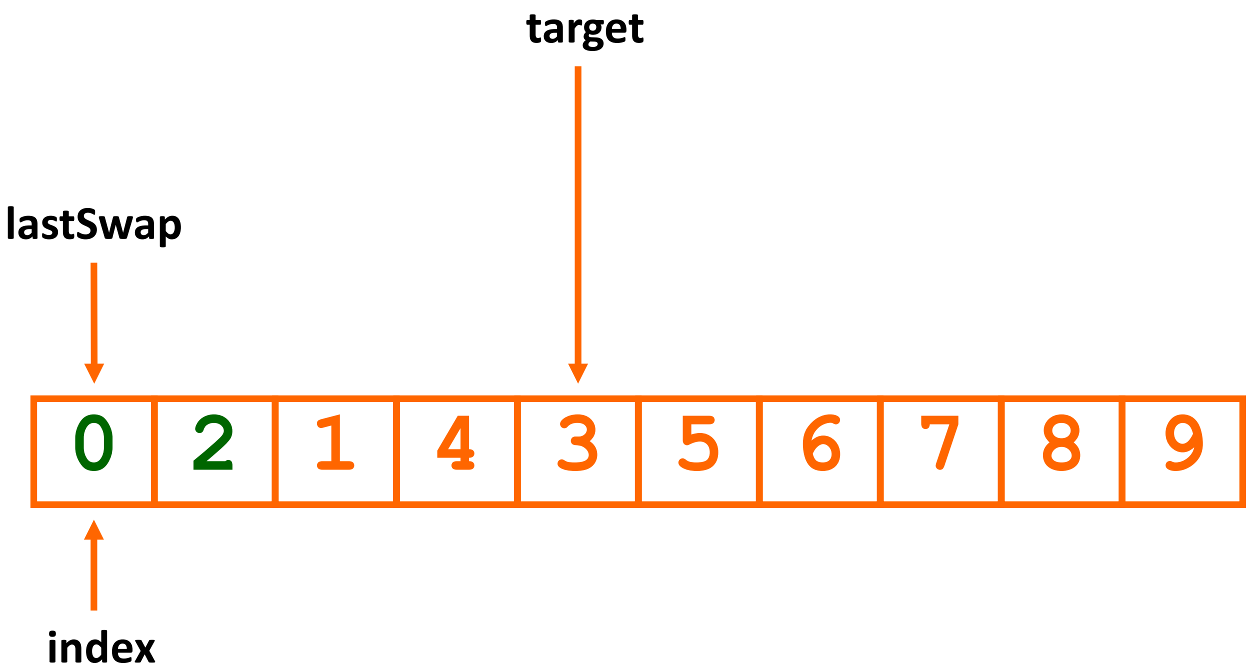
Bubble Sort Animation



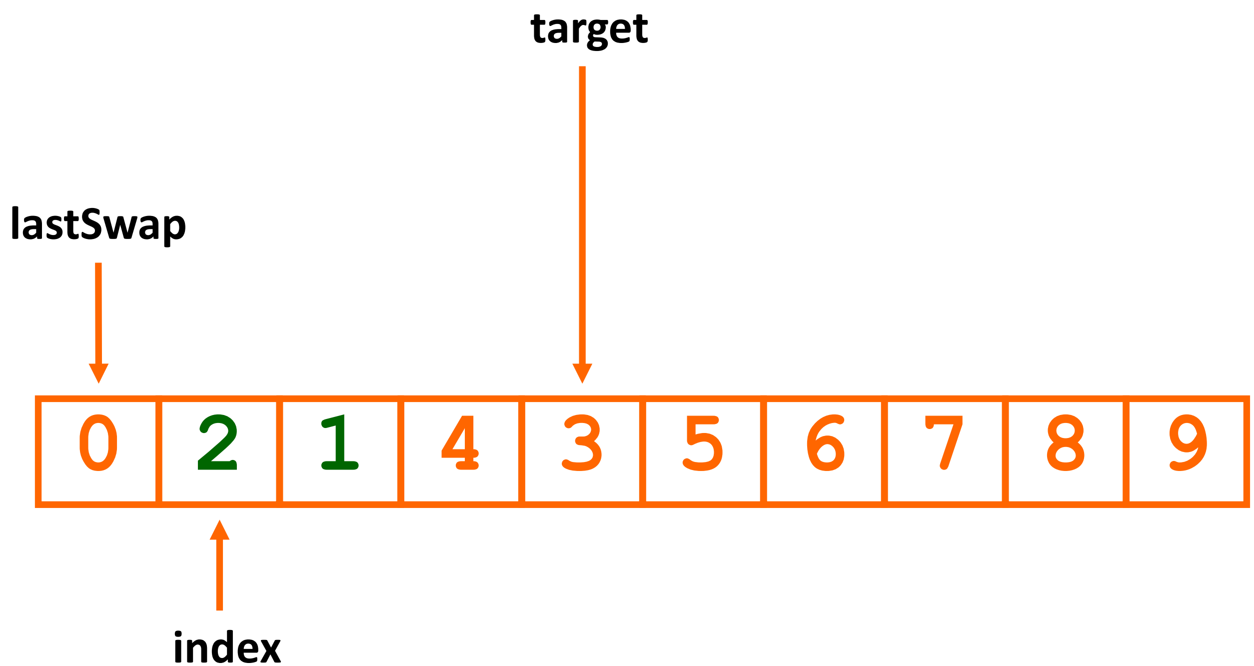
Bubble Sort Animation



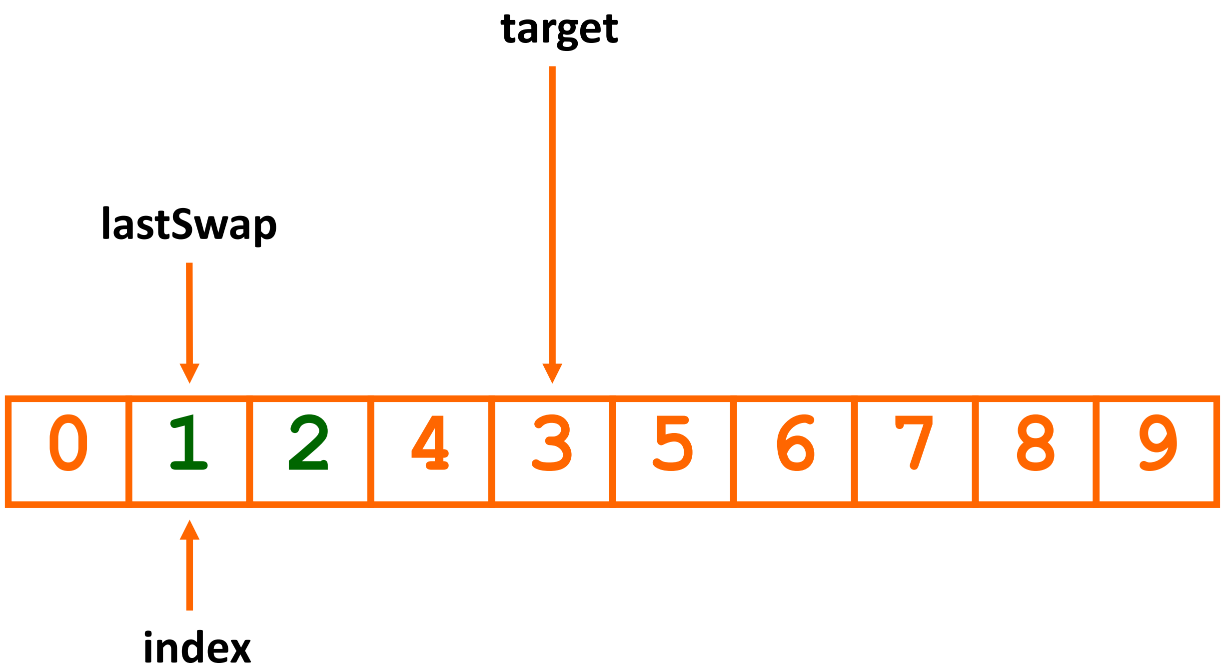
Bubble Sort Animation



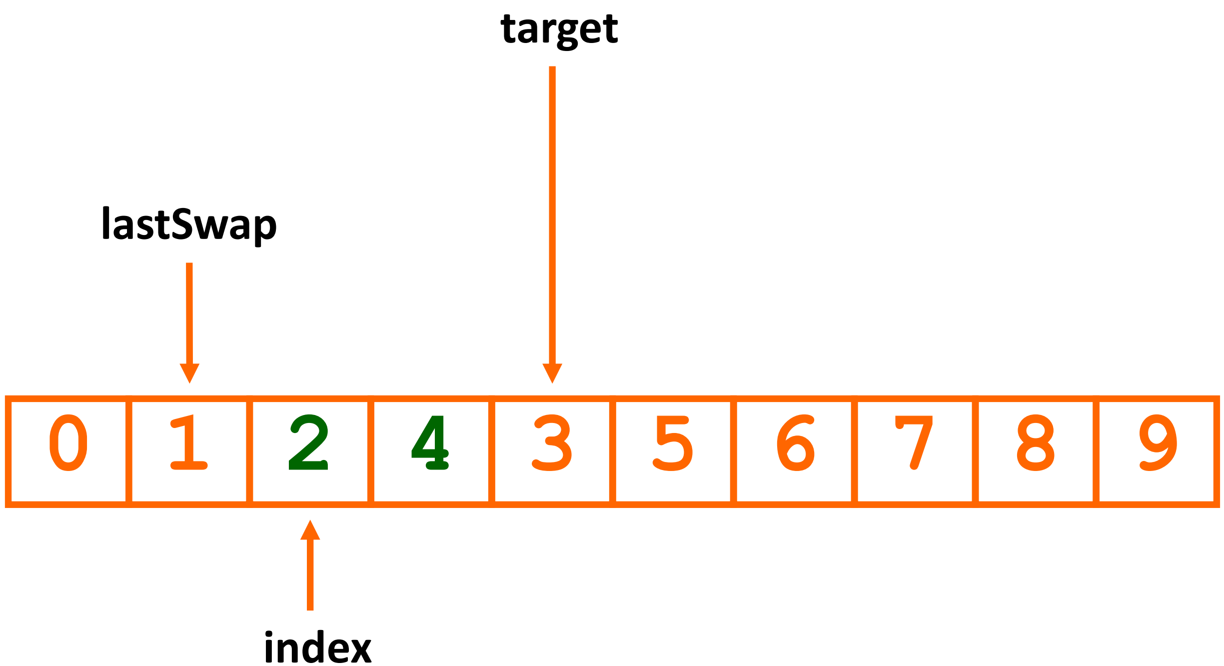
Bubble Sort Animation



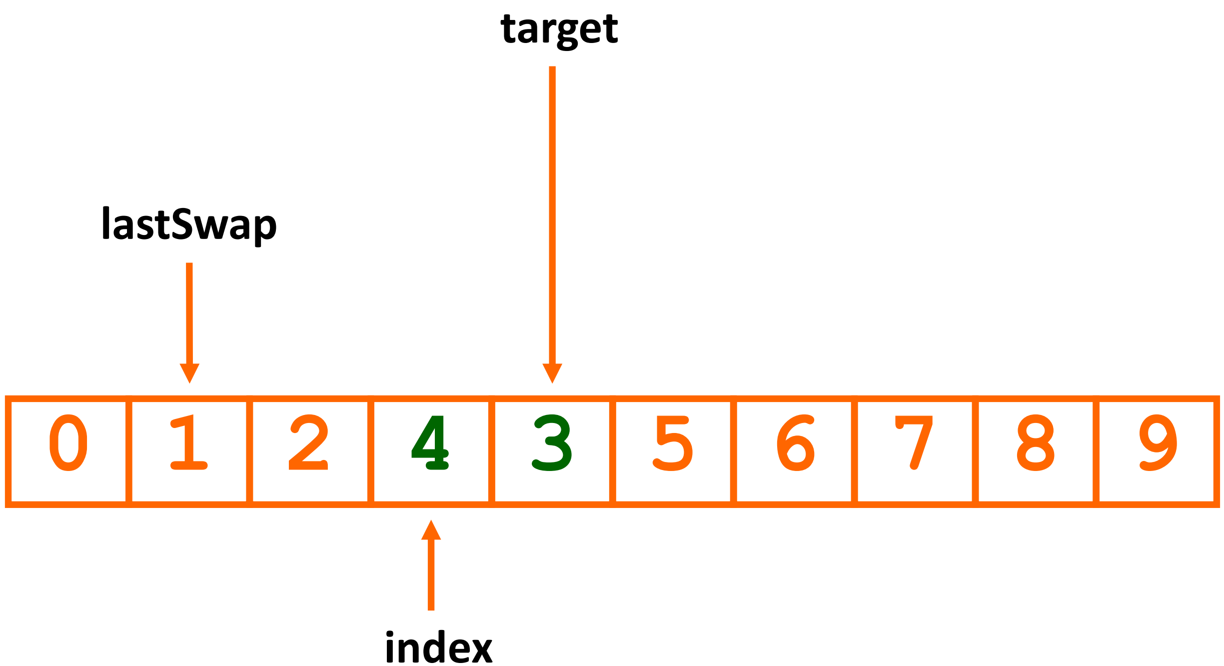
Bubble Sort Animation



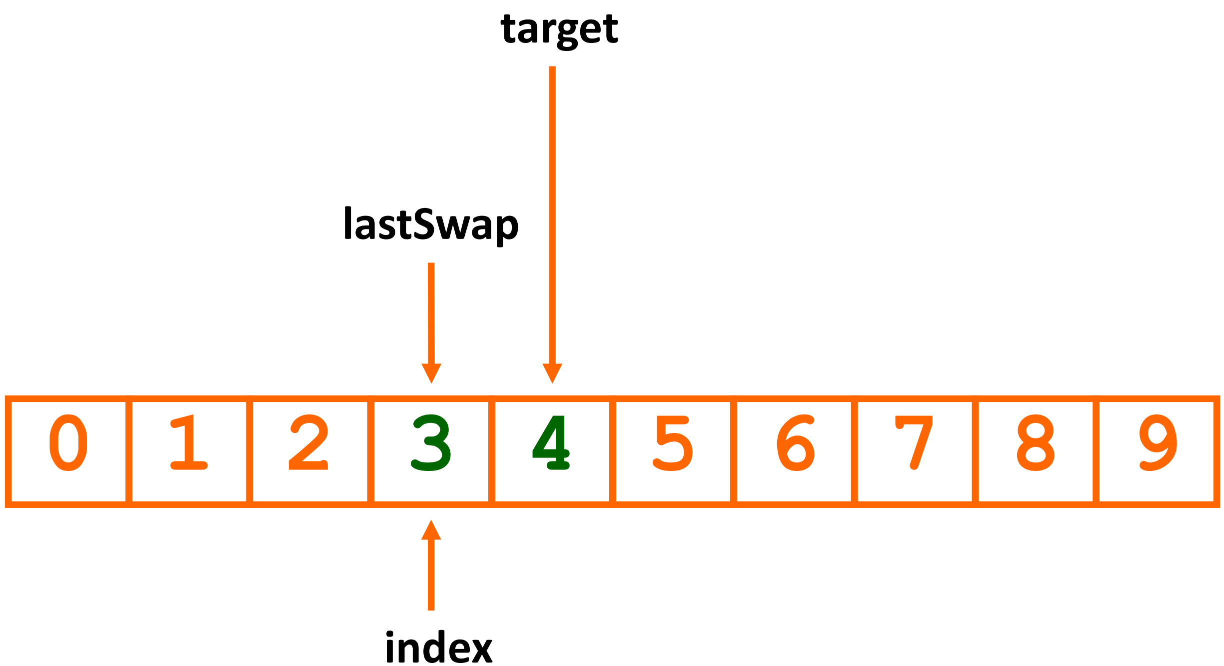
Bubble Sort Animation



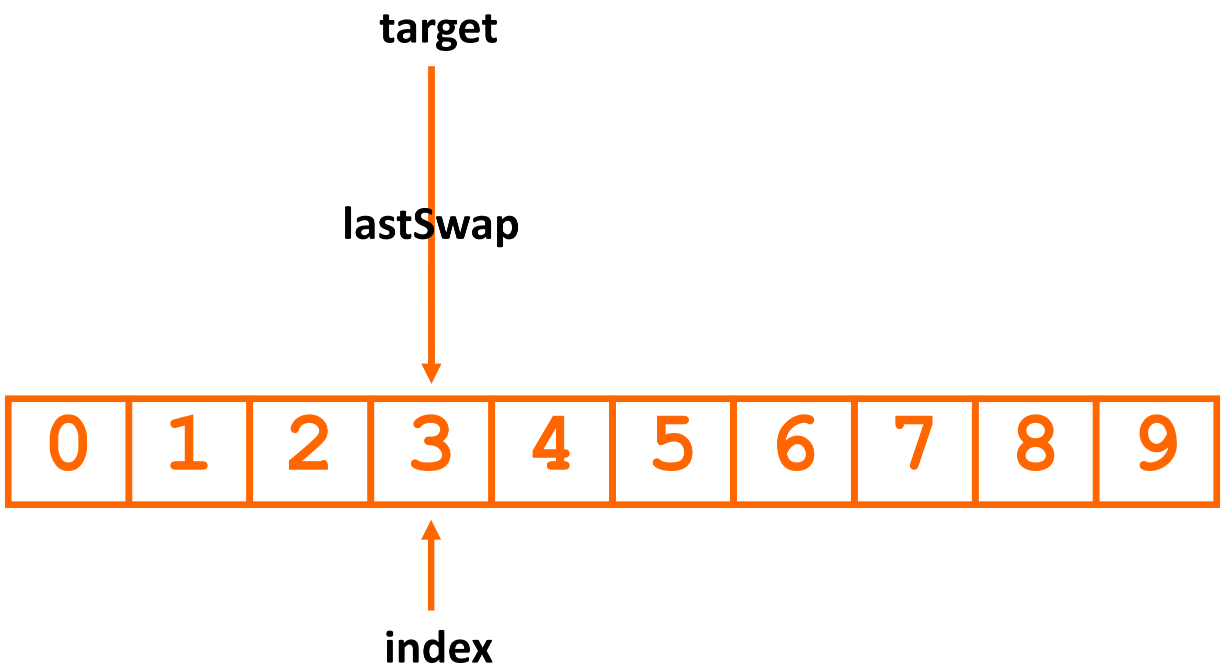
Bubble Sort Animation



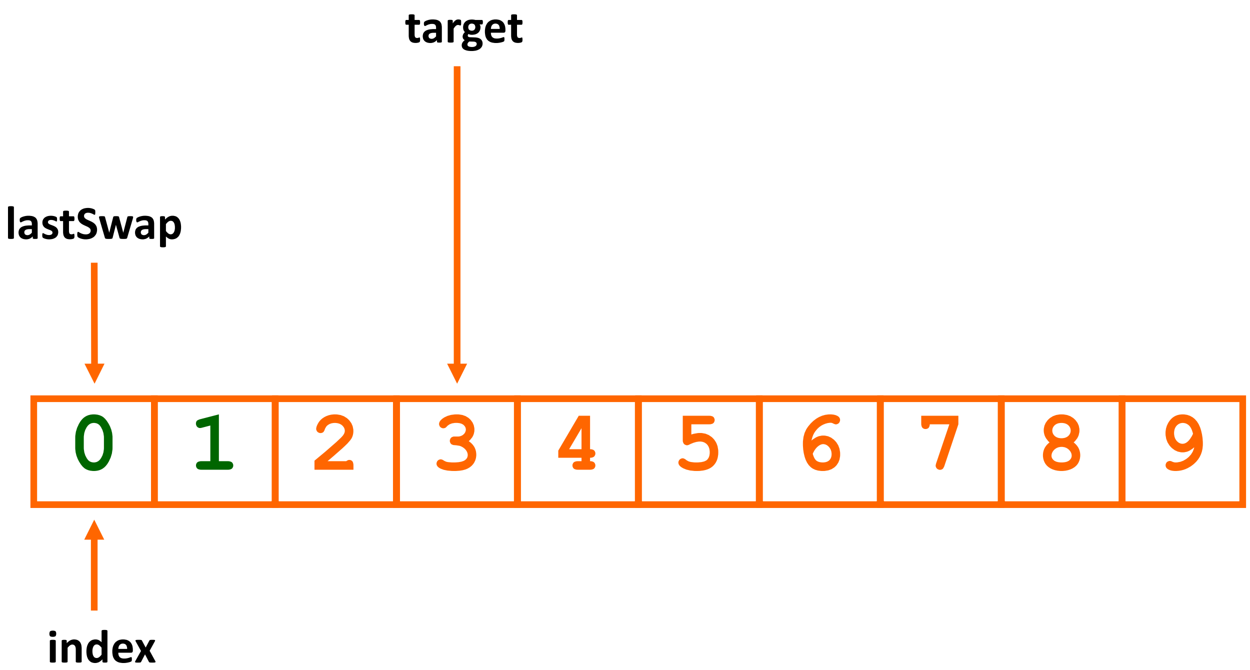
Bubble Sort Animation



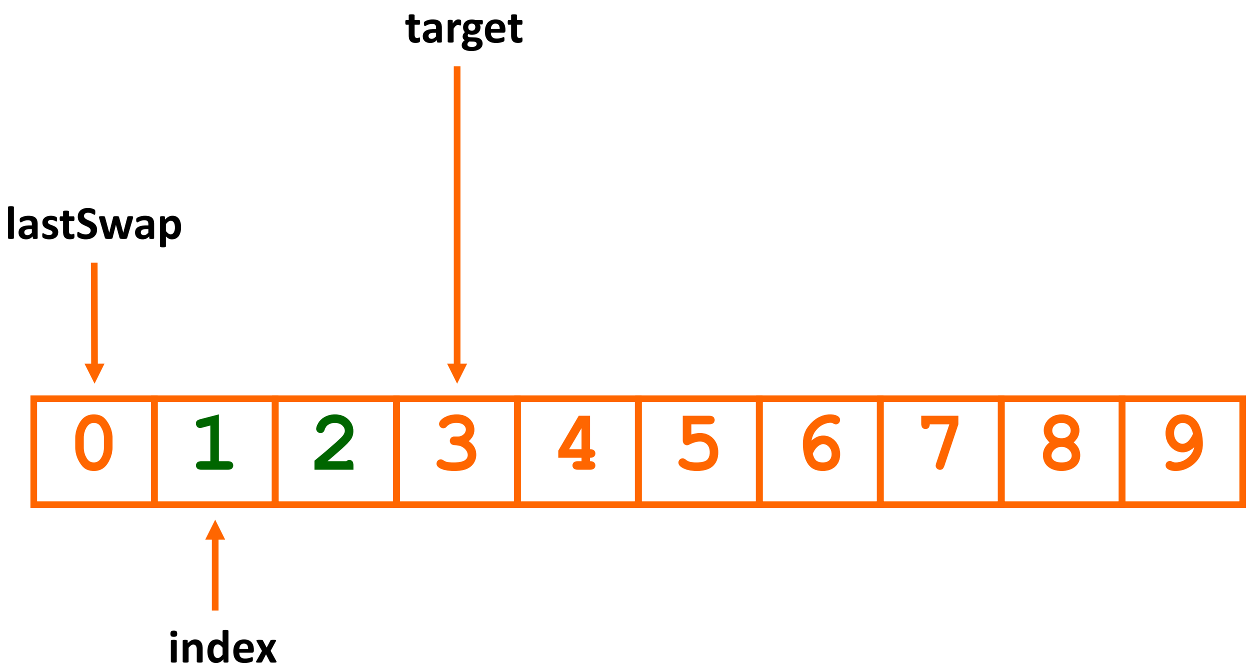
Bubble Sort Animation



Bubble Sort Animation

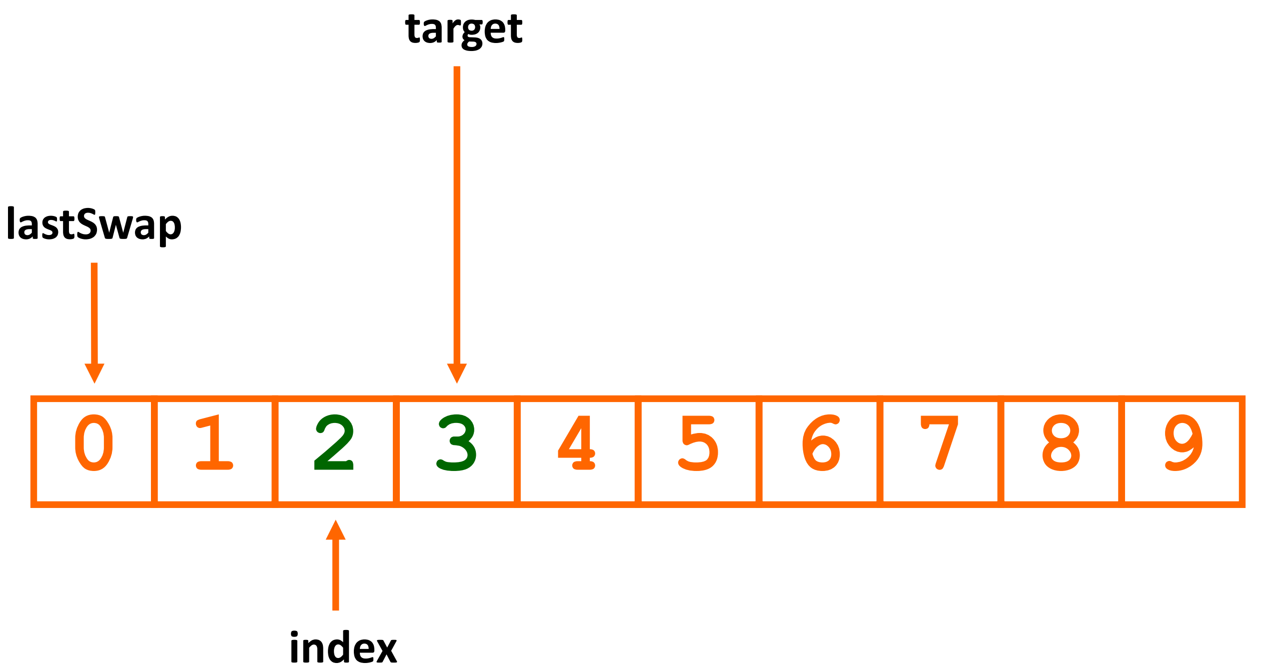


Bubble Sort Animation



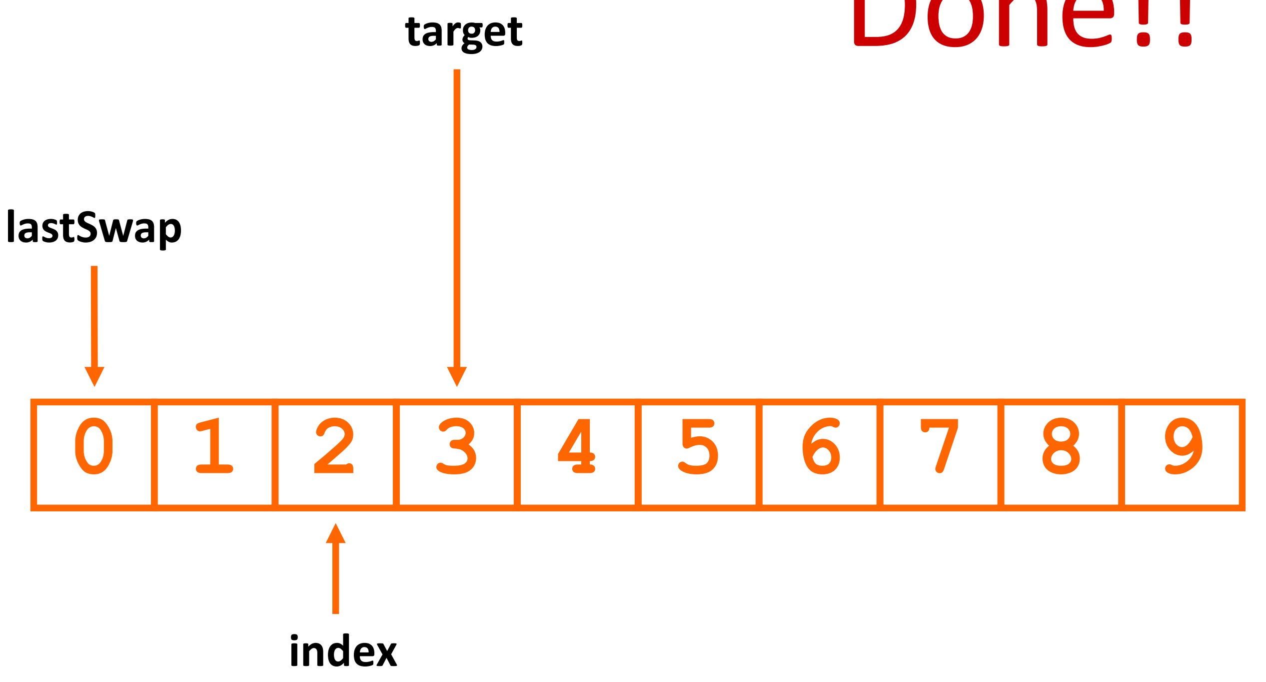
Murdoch UNIVERSITY

Bubble Sort Animation



Bubble Sort Animation

Done!!



Merge Sort [1]

- Merge sort uses the divide and conquer algorithmic strategy.
- It has complexity O(nlog n) for all cases.
- It is a simple merge to implement.
- It is most easily implemented using recursion.
- It is an efficient sort to implement for a large amount of data on disk (that does not fit into RAM).

Murdoch UNIVERSITY

Merge Sort Algorithm

```
•   MergeSort

•       IF there are more than two elements in the container
•           Divide the container into two
•           Merge Sort the first part // call again
•           Merge Sort the second part // call again
•           Merge the two sorted parts into a temp file or
                    array
•           Put merged temp file/array back into array being
                        sorted
•       ELSE IF two elements in the container
•           Swap them if necessary
•       ENDIF

•   END MergeSort
```

Merge Sort Animation

2	9	0	7	8	6	5	1	4	3

Merge Sort Animation

2	9	0	7	8	6	5	1	4	3

Murdoch UNIVERSITY

Merge Sort Animation

2	9	0	7	8	6	5	1	4	3

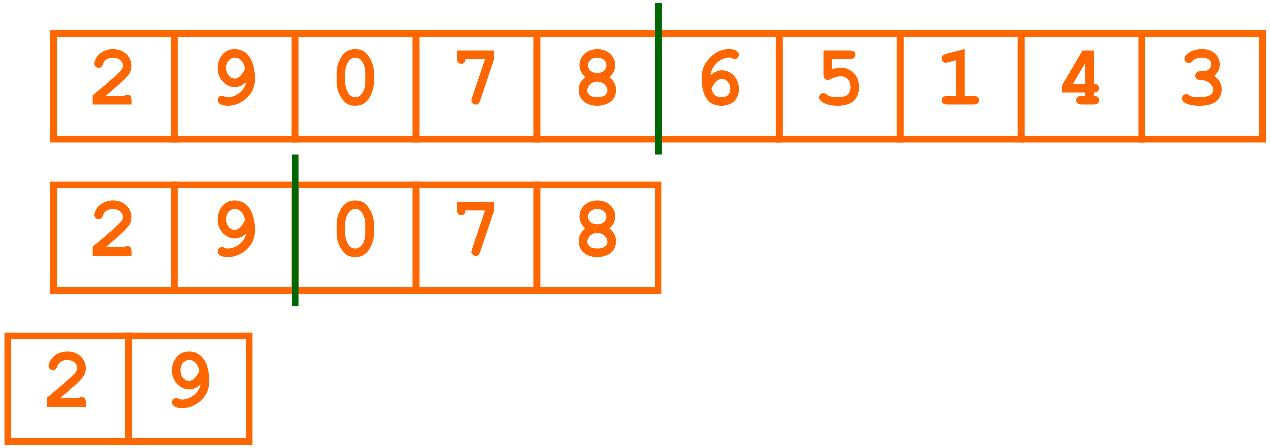
2	9	0	7	8

Merge Sort Animation

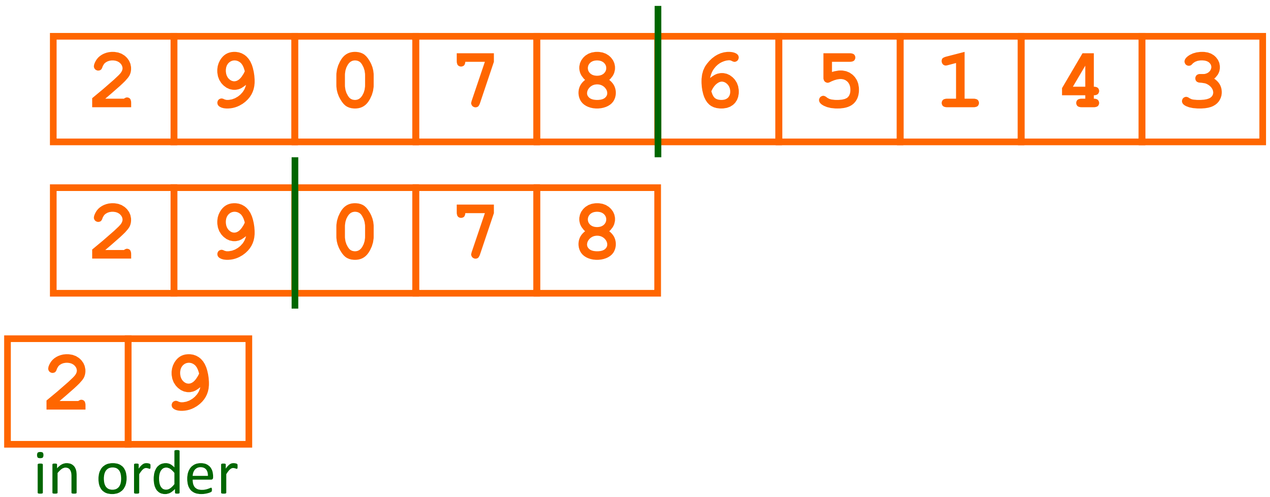
2	9	0	7	8	6	5	1	4	3

2	9	0	7	8

Merge Sort Animation



Merge Sort Animation



Merge Sort Animation

2	9	0	7	8	6	5	1	4	3

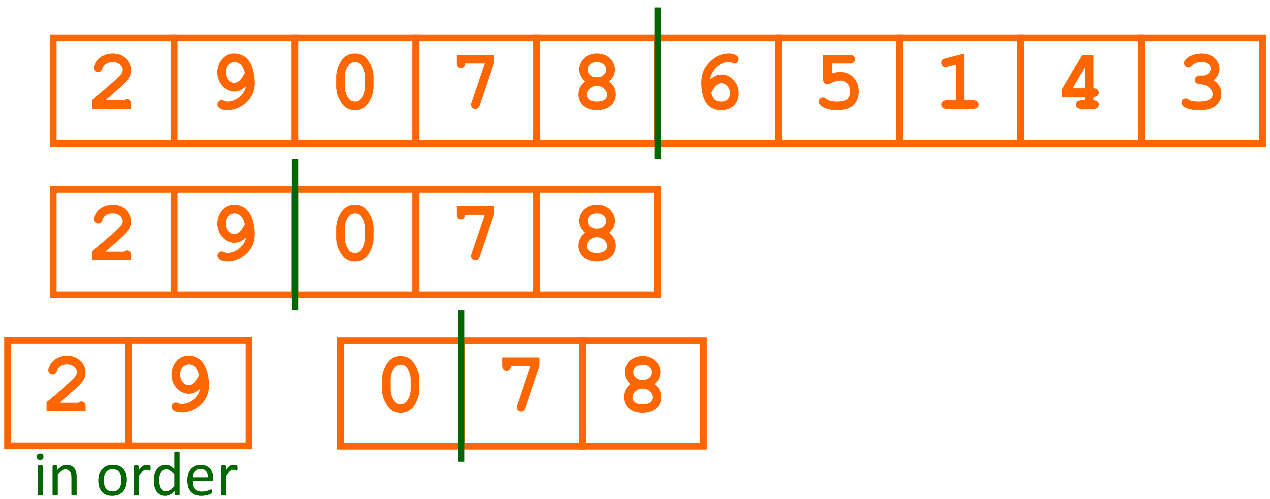
2	9	0	7	8

2	9

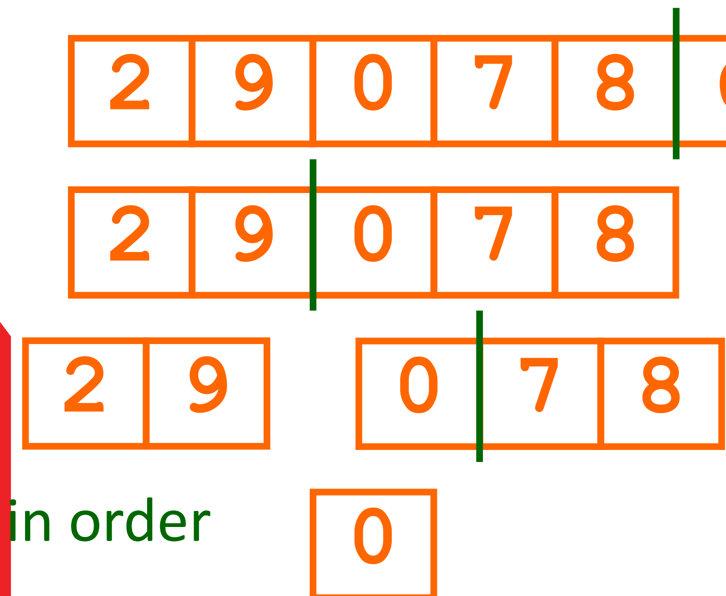
in order

0	7	8

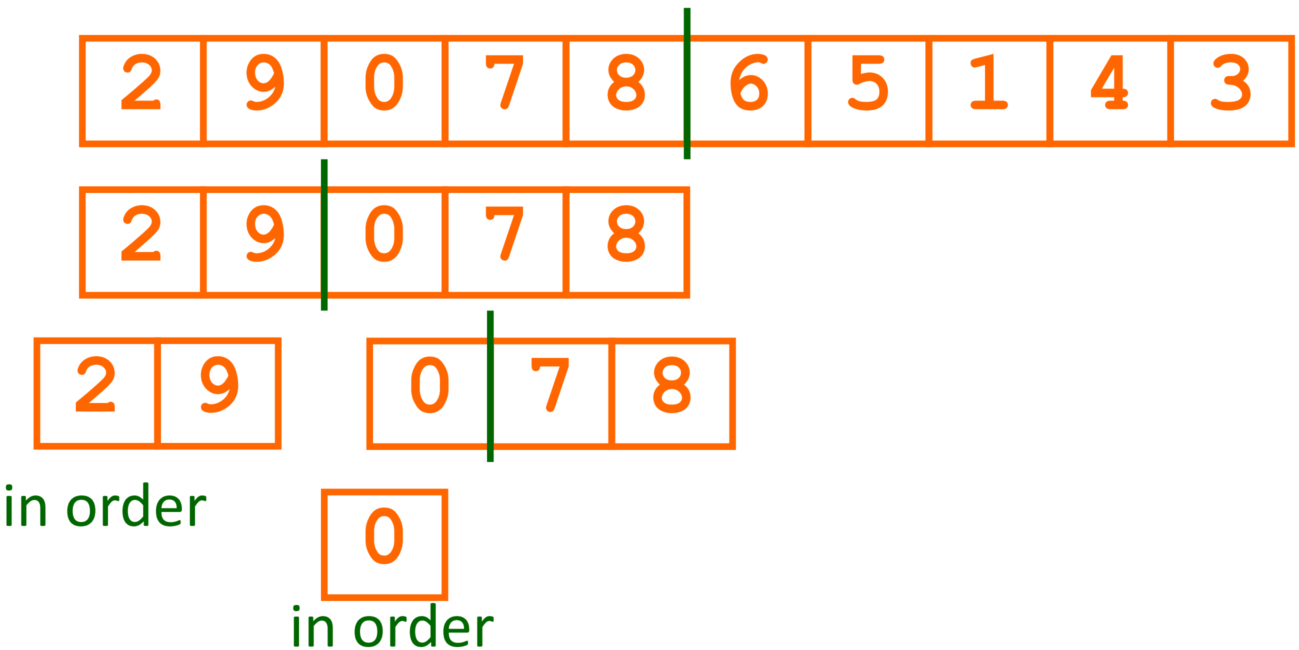
Merge Sort Animation



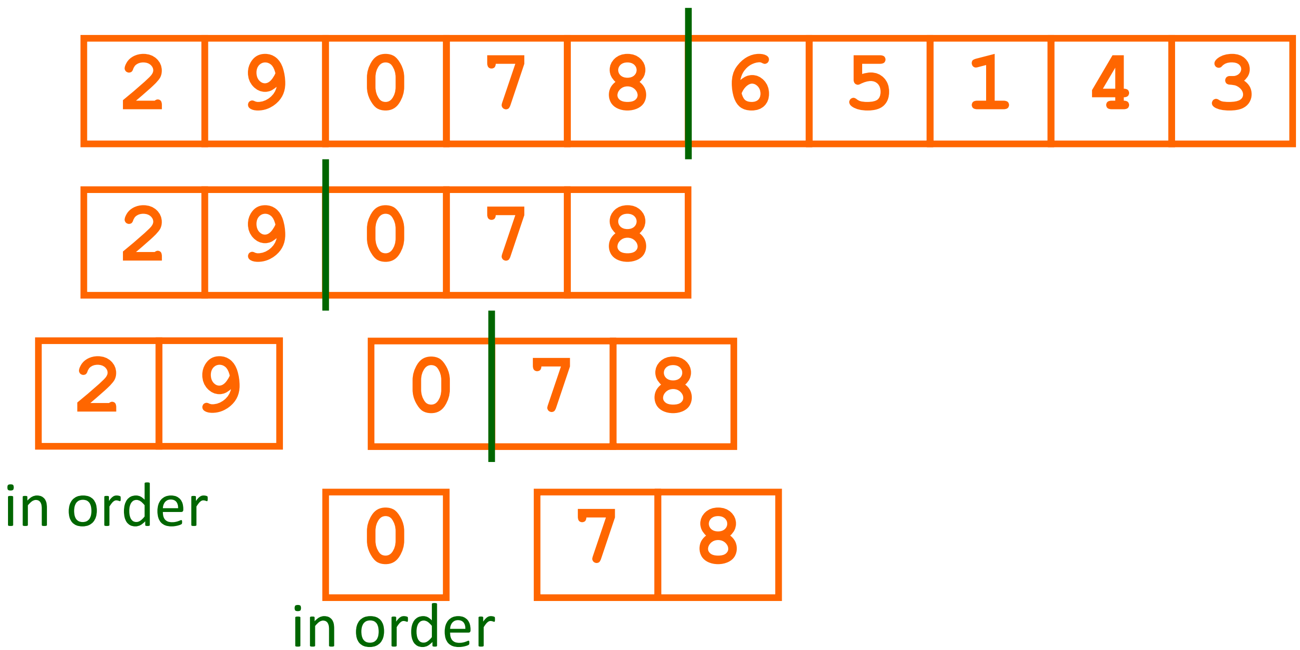
Merge Sort Animation



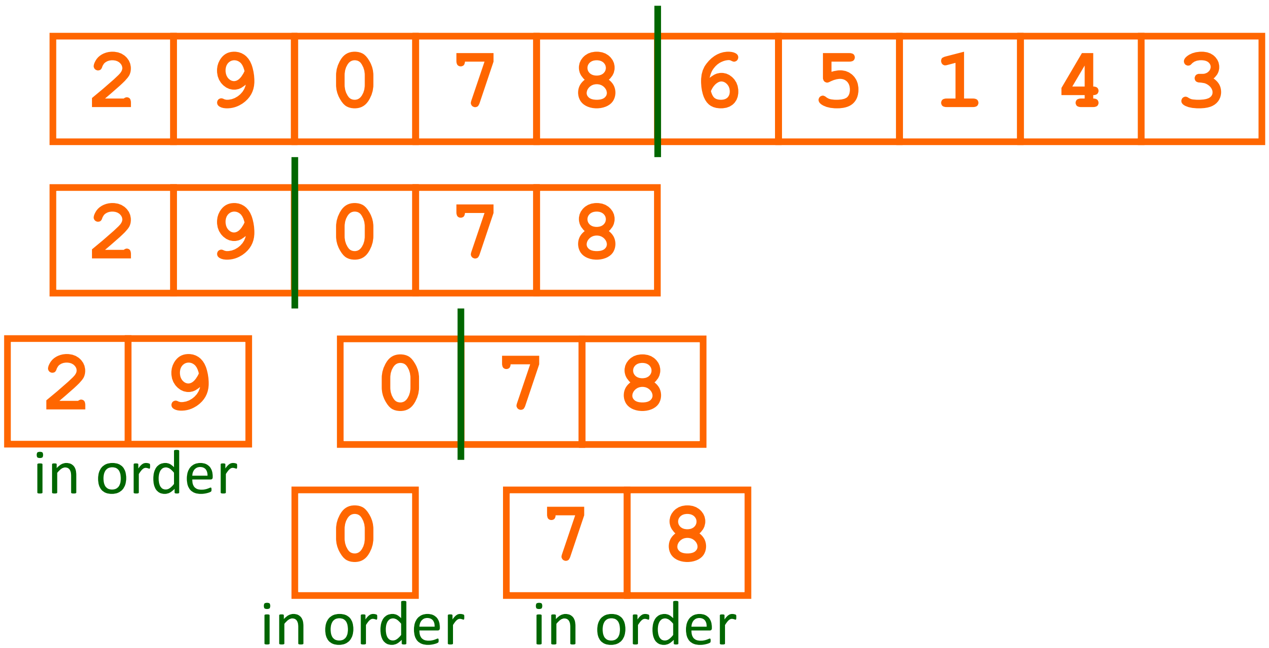
Merge Sort Animation



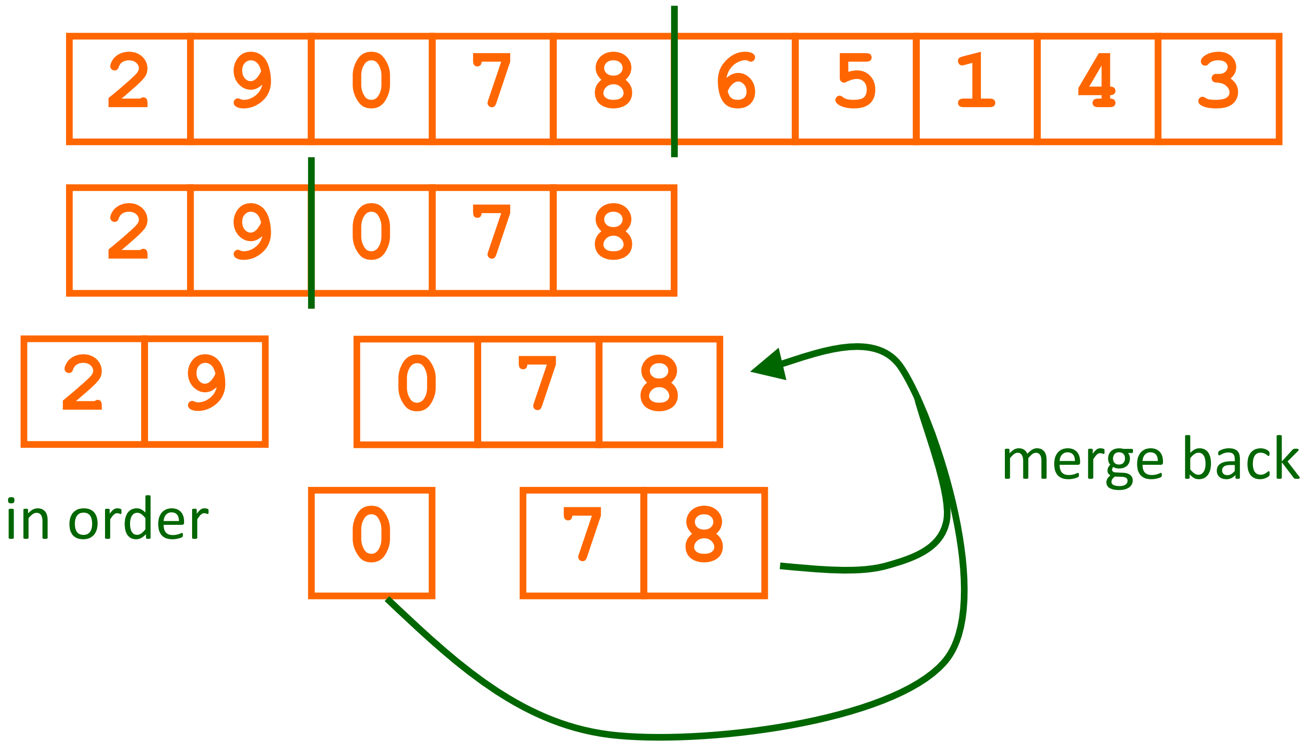
Merge Sort Animation



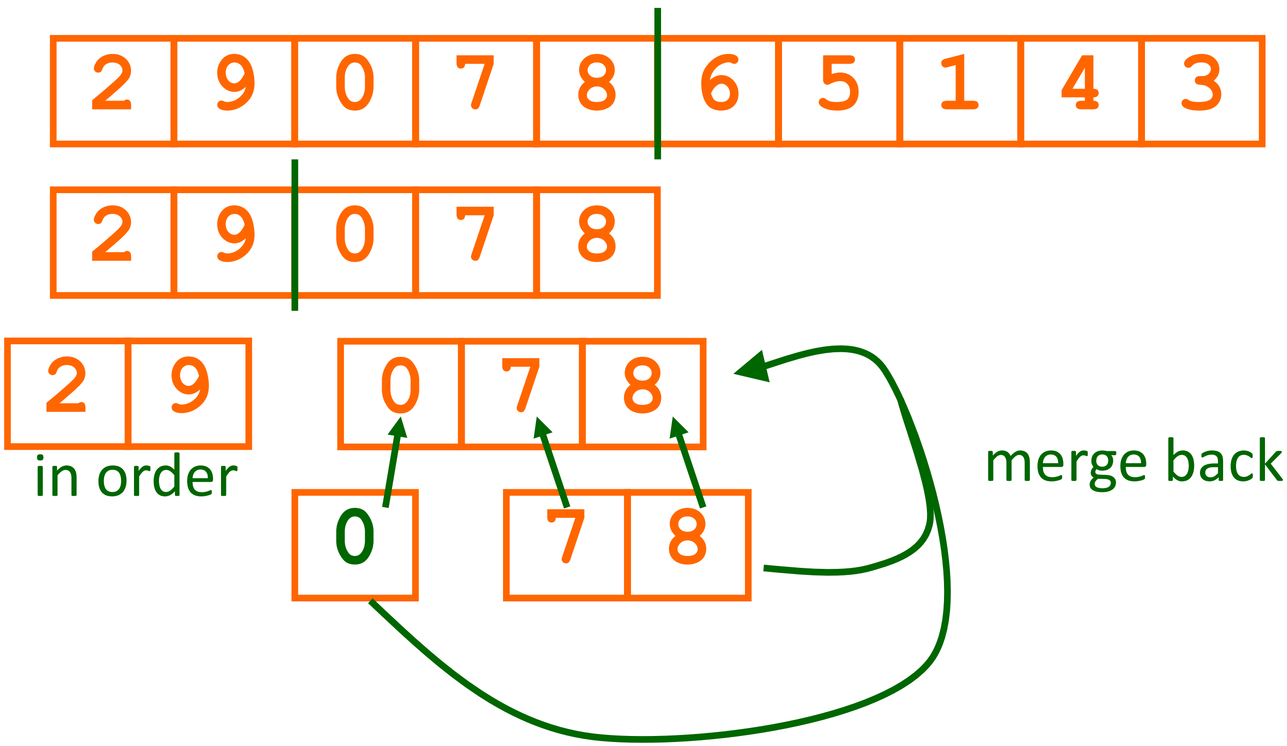
Merge Sort Animation



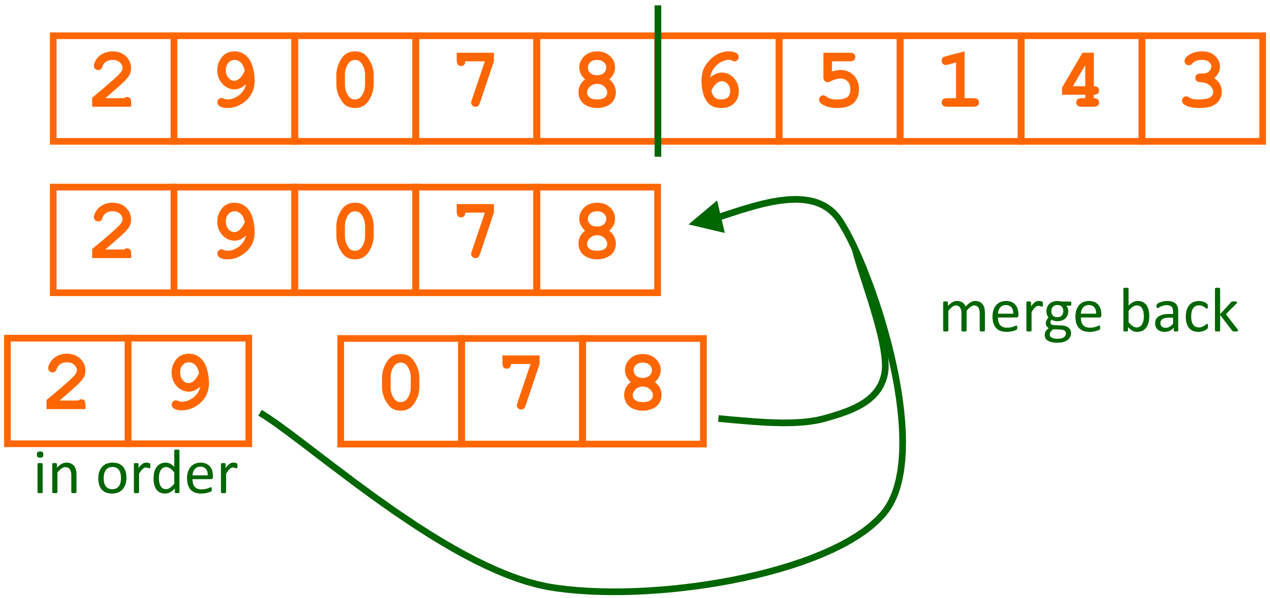
Merge Sort Animation



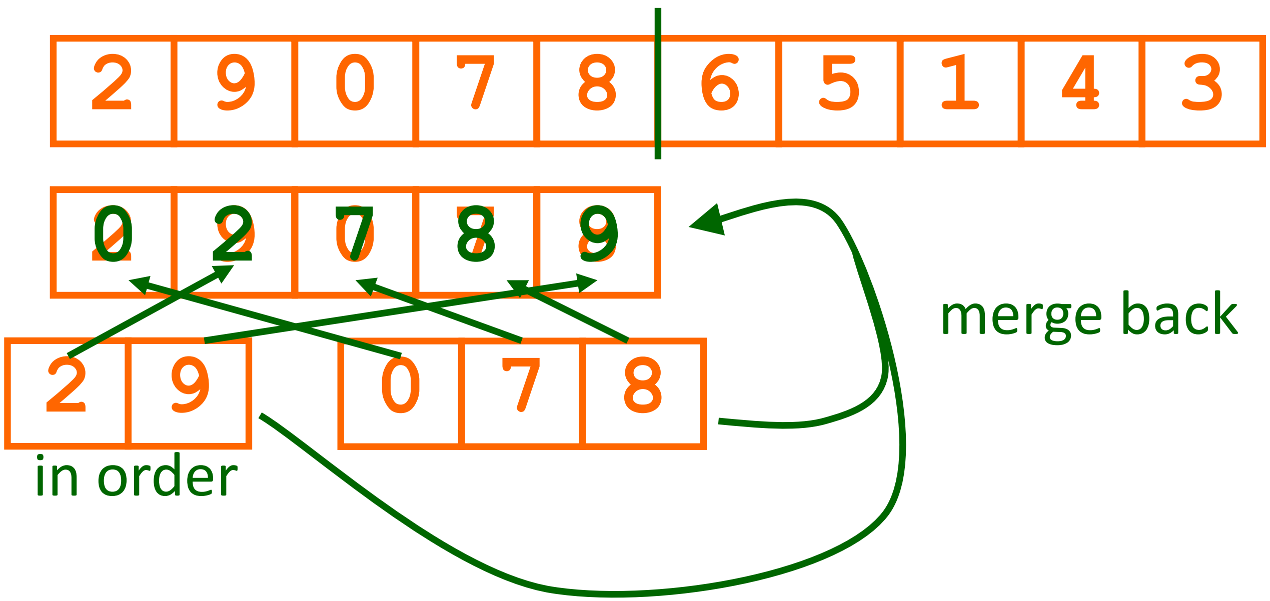
Merge Sort Animation



Merge Sort Animation



Merge Sort Animation



Merge Sort Animation

2	9	0	7	8	6	5	1	4	3

0	2	7	8	9

Merge Sort Animation

2	9	0	7	8	6	5	1	4	3

0	2	7	8	9

6	5	1	4	3

Murdoch UNIVERSITY

Merge Sort Animation

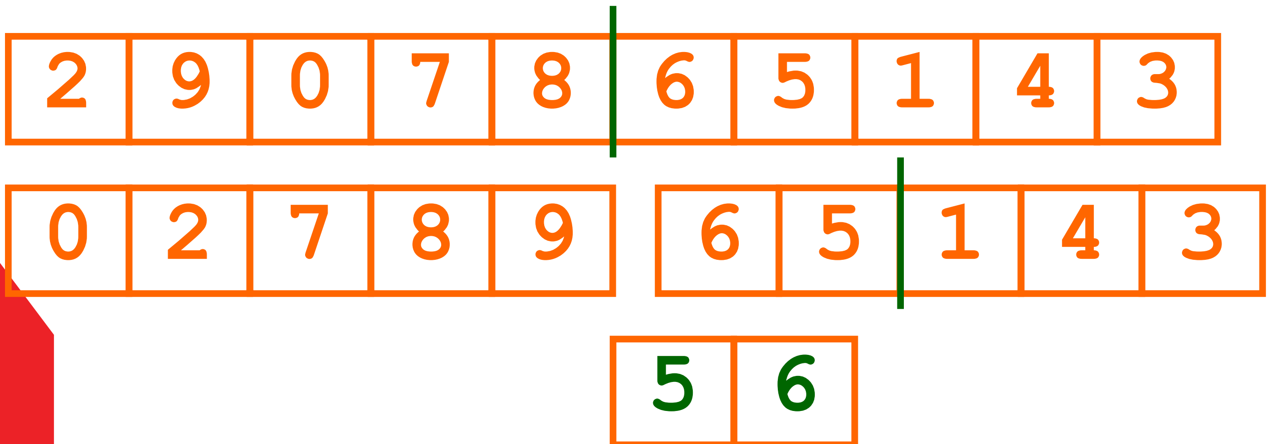
2	9	0	7	8	6	5	1	4	3

0	2	7	8	9

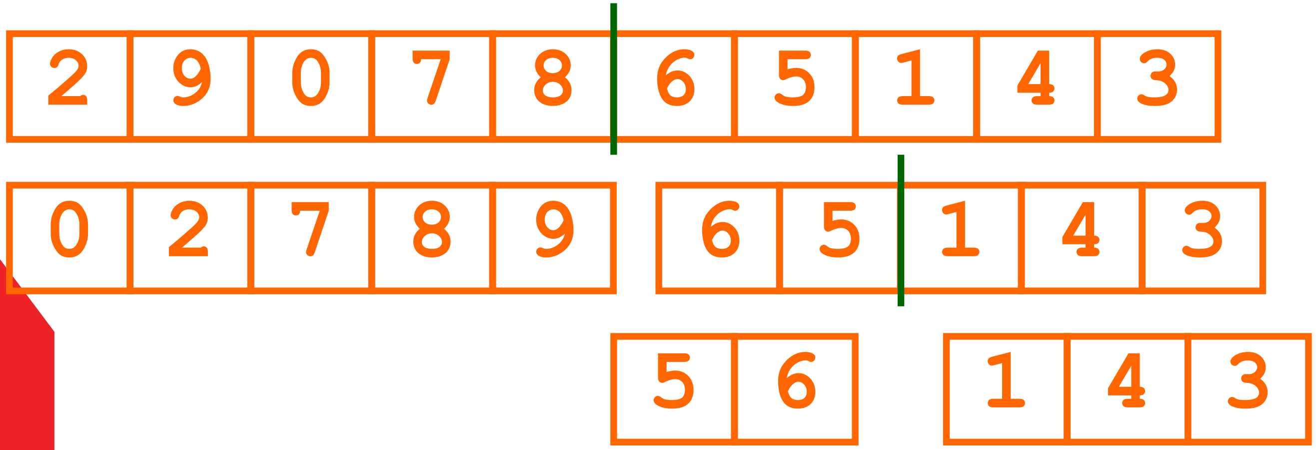
6	5	1	4	3

6	5

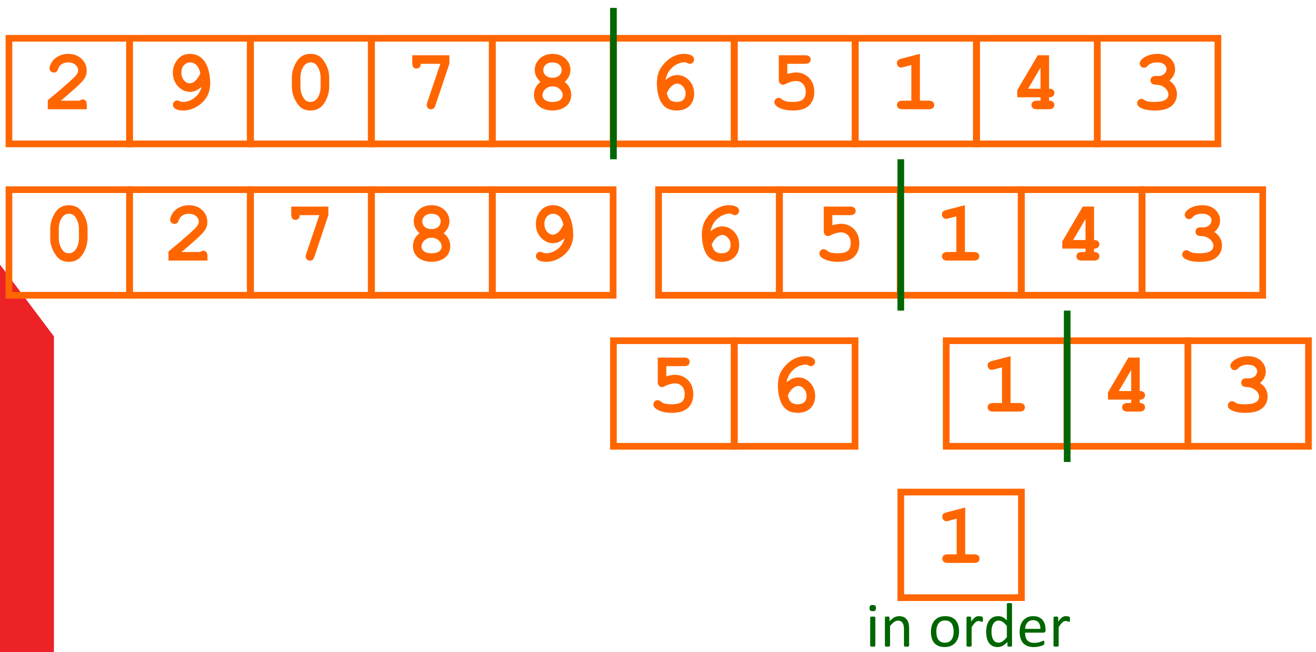
Merge Sort Animation



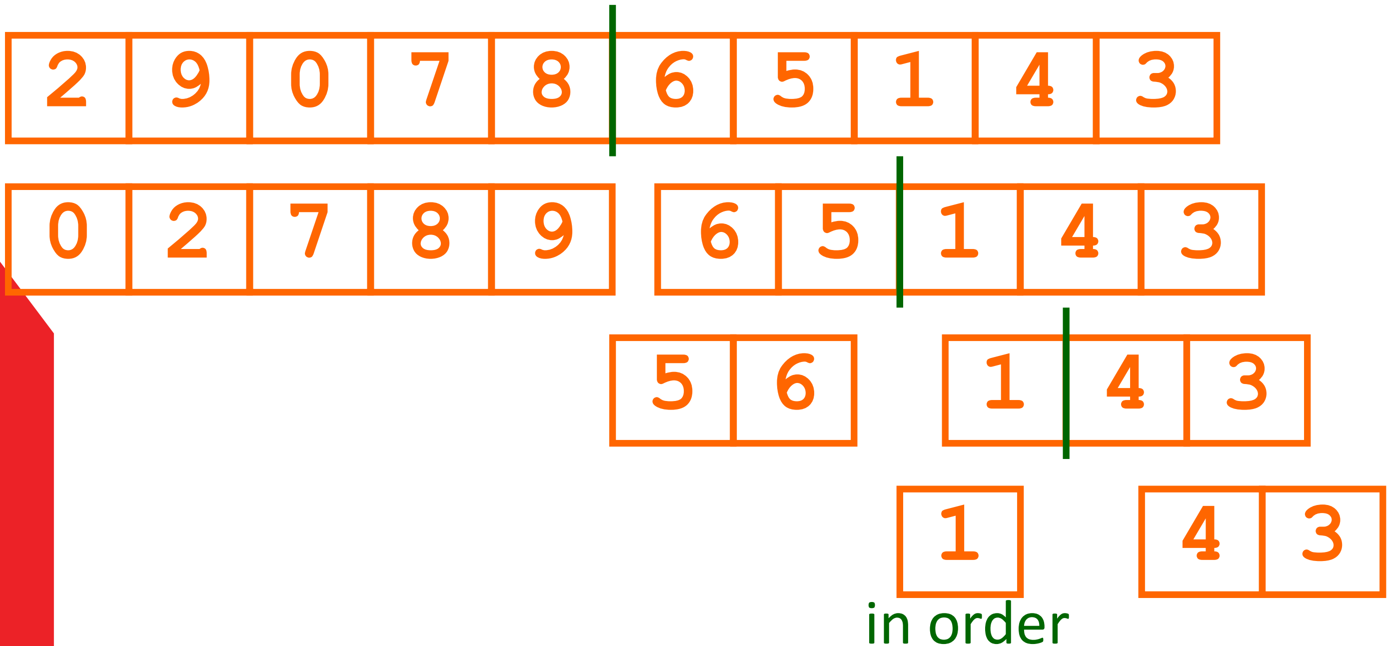
Merge Sort Animation



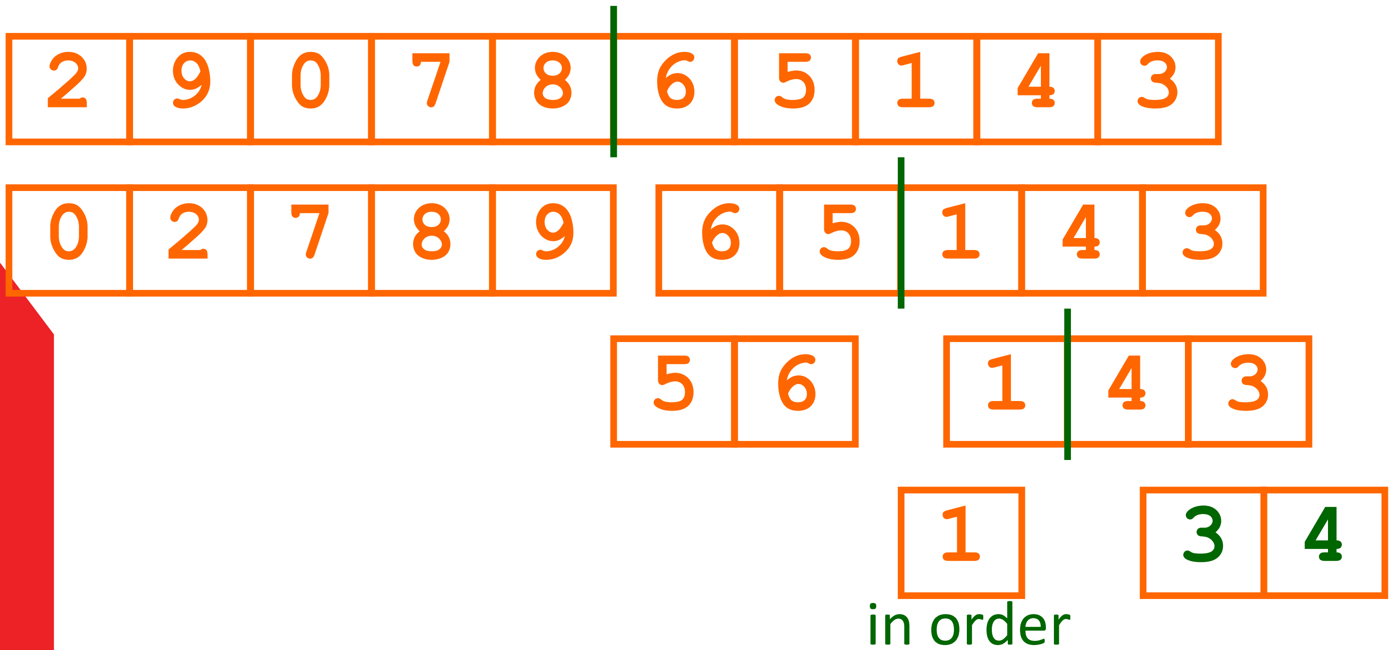
Merge Sort Animation



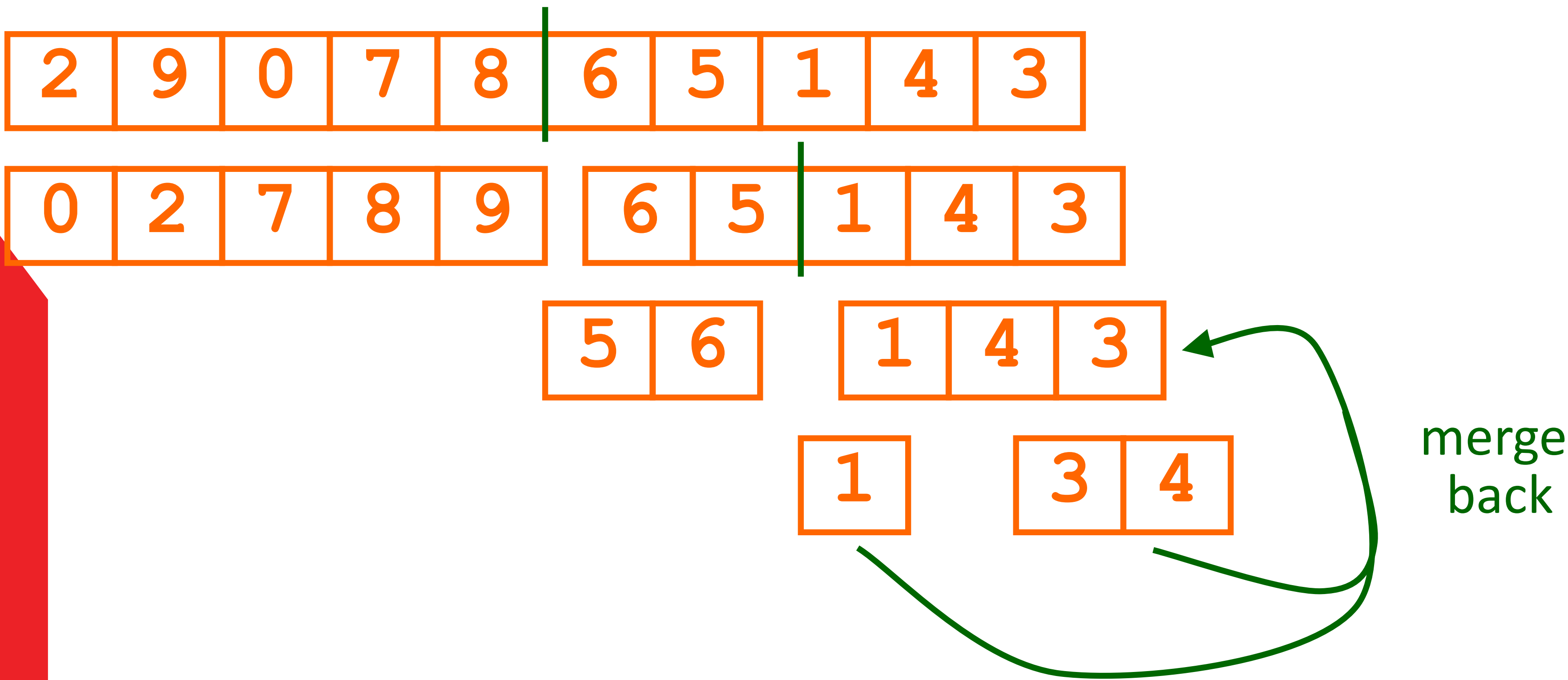
Merge Sort Animation



Merge Sort Animation

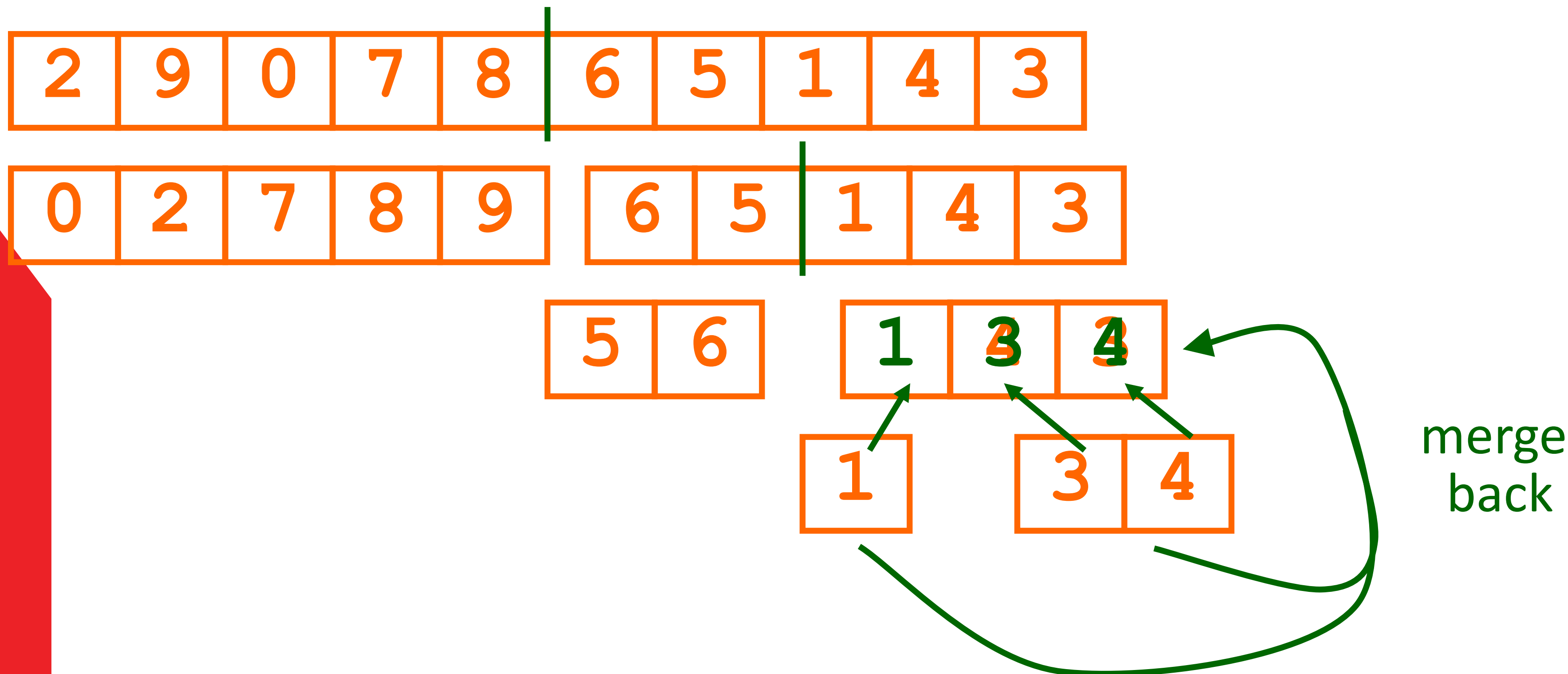


Merge Sort Animation

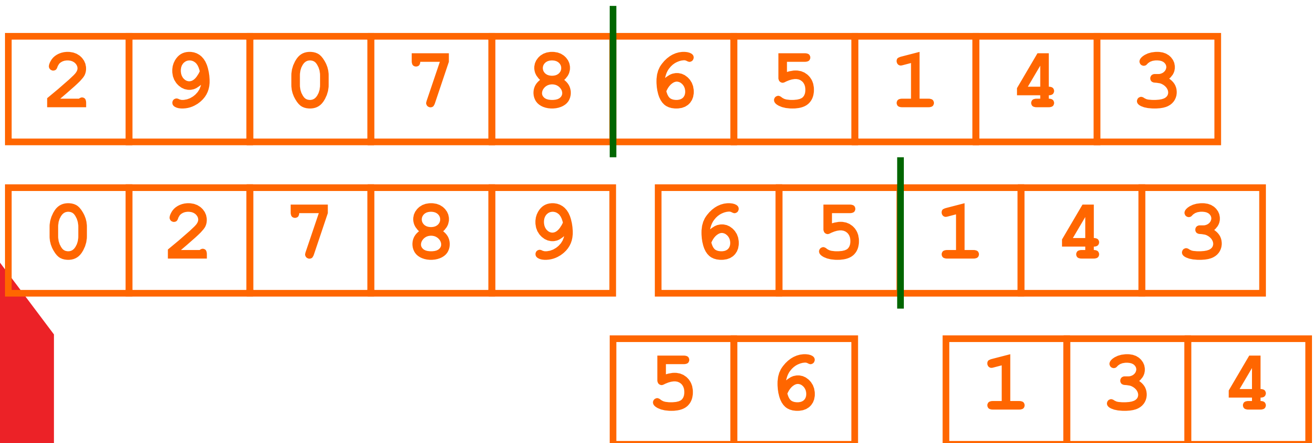


Murdoch UNIVERSITY

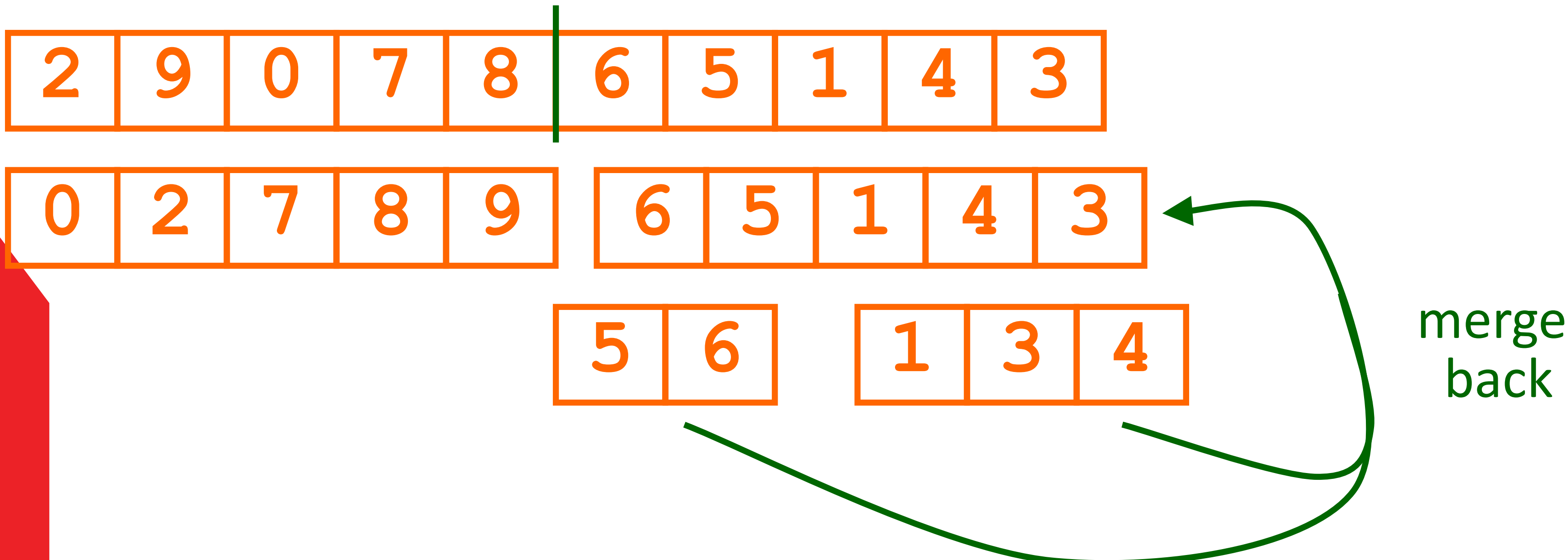
Merge Sort Animation



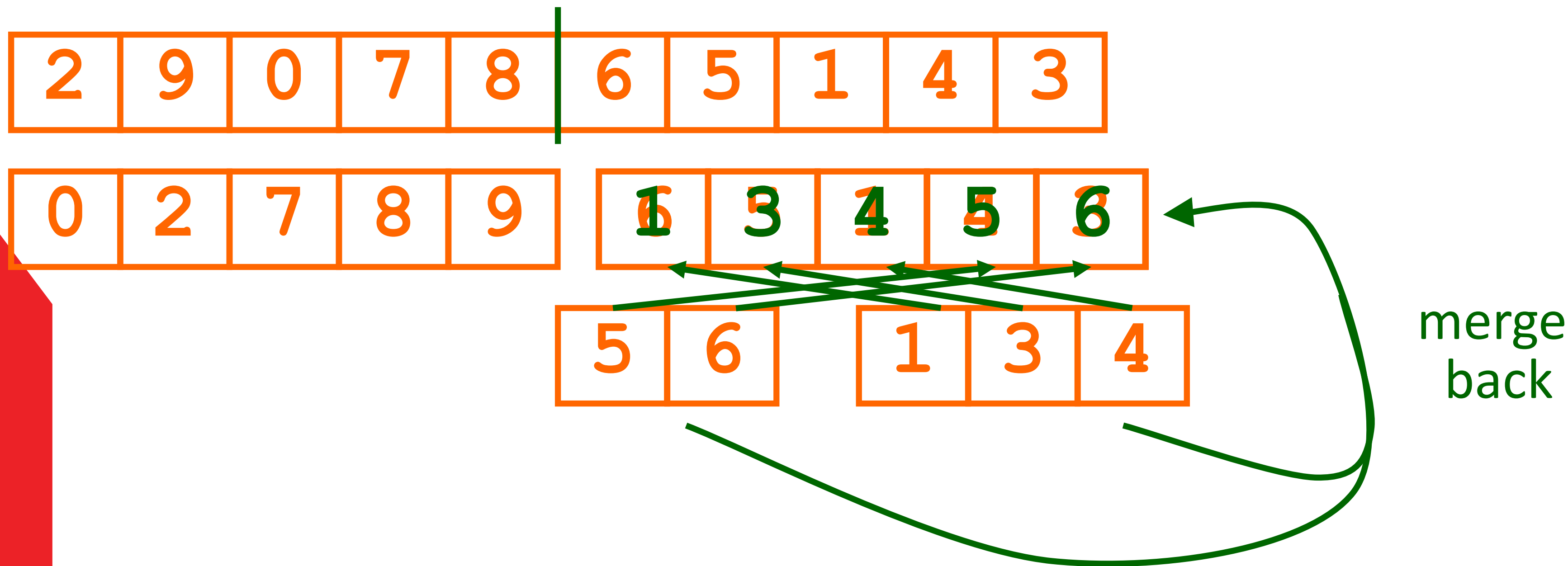
Merge Sort Animation



Merge Sort Animation



Merge Sort Animation



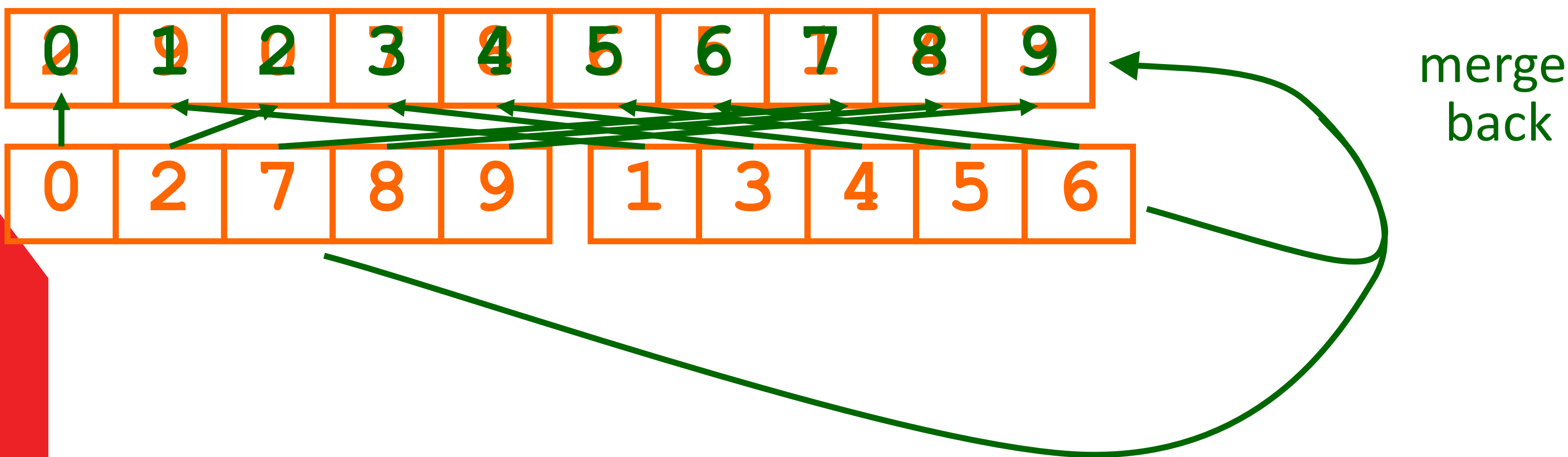
Merge Sort Animation

2	9	0	7	8	6	5	1	4	3

0	2	7	8	9

1	3	4	5	6

Merge Sort Animation



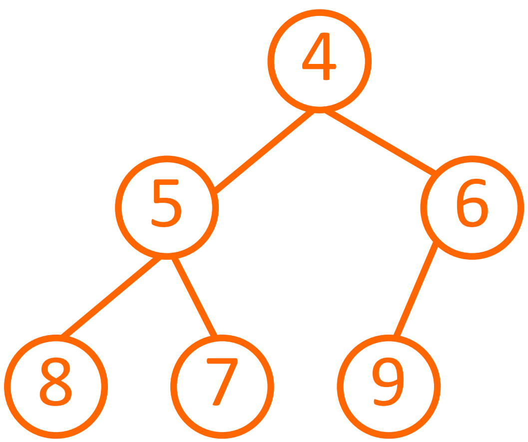
Merge Sort Animation

0	1	2	3	4	5	6	7	8	9

Done!!

The Abstract Heap

- A ‘heap’ is a data structure in the form of a binary tree.
- A binary tree is one where each node contains data plus 0-2 pointers to nodes underneath it.



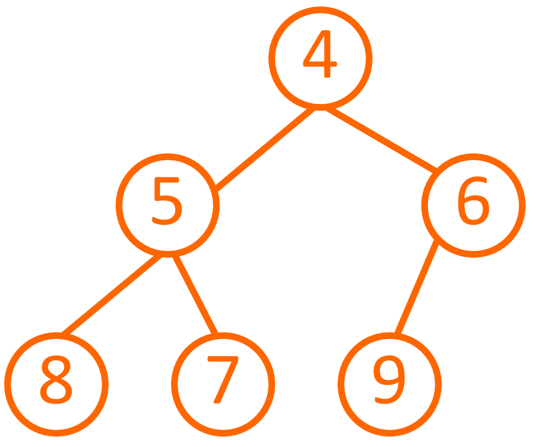
- A heap is a binary tree where the data in a node is guaranteed to be either less than (a min heap) or greater than (a max heap) all of the data below it.

Murdoch UNIVERSITY

The Actual Heap

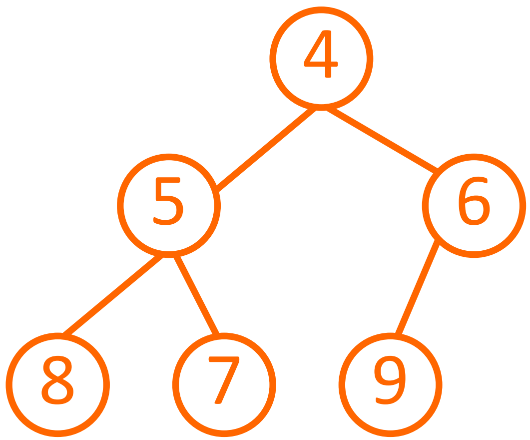
- Clearly such a structure can be used to sort data.
- However, in actual fact, the data structure used is simply another array.
- This is because we end up doing a lot of data swapping in a heap, which is difficult to code in an actual tree.
- Also it turns out that in an array, the parent-child relationships is mathematical, making swaps particularly easy.

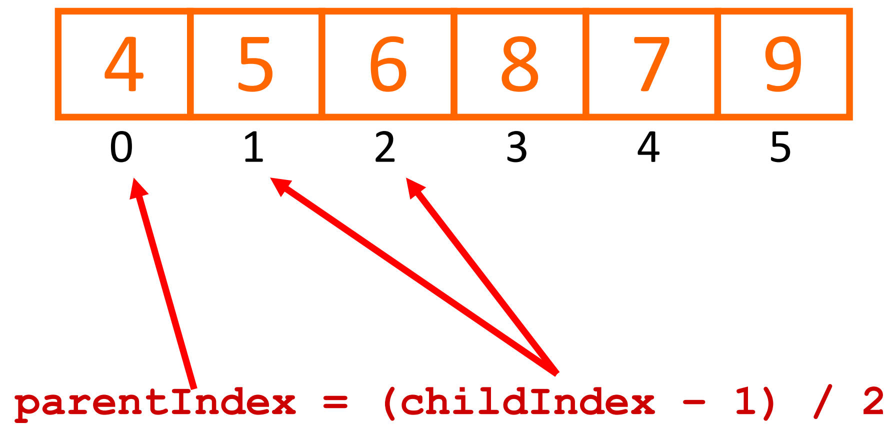
Abstract View vs Actual View



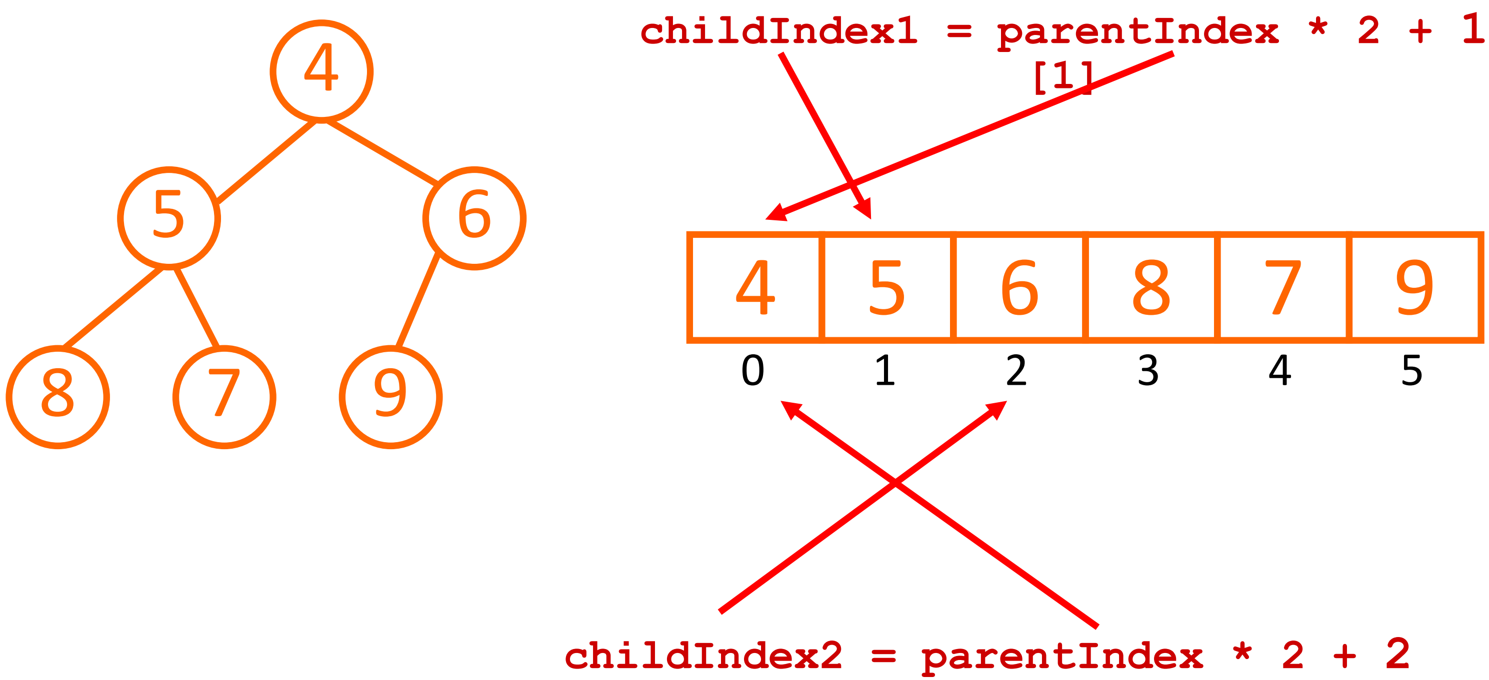
4	5	6	8	7	9

Abstract View vs Actual View





Abstract View vs Actual View



Murdoch UNIVERSITY

Heap Sort Algorithm

- Heap sort is an unstable selection sort.
- It utilises a greedy algorithmic technique.
- It has complexity O(nlog n).
- But is more complicated to code than a merge sort.

Murdoch UNIVERSITY

Heap Sort Algorithm

```
•   HeapSort

•       FOR each member of the array
•           Place it at the bottom end of the heap
•           WHILE it is smaller than the parent
•               Exchange it with the parent
•           ENDWHILE
•       ENDFOR

•       index = 0
•       WHILE the heap is not empty
•           Put the top of the heap at index in the array
•           Increment index
•           Delete the top of the heap and rearrange
•       ENDWHILE

•   END HeapSort
```

Put on the heap

Take off the heap

Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

Abstract Heap

Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

2									

Abstract Heap

2

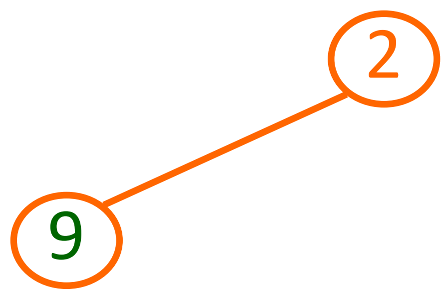
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

2	9								

Abstract Heap



Inserts go at left-most bottom row

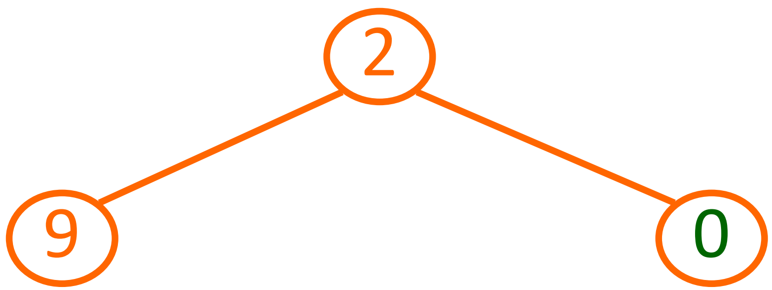
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

2	9	0							

Abstract Heap



Inserts go at left-most bottom row

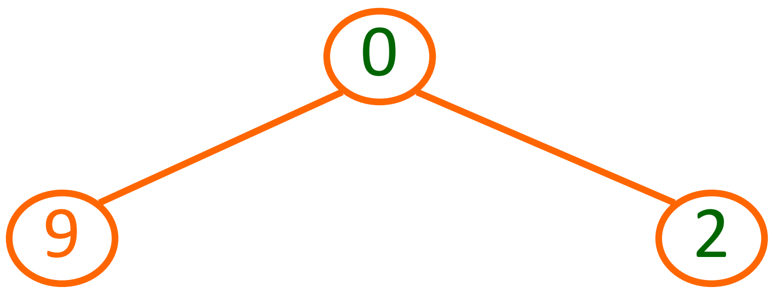
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	9	2							

Abstract Heap



Values then
‘percolate’ up
if smaller

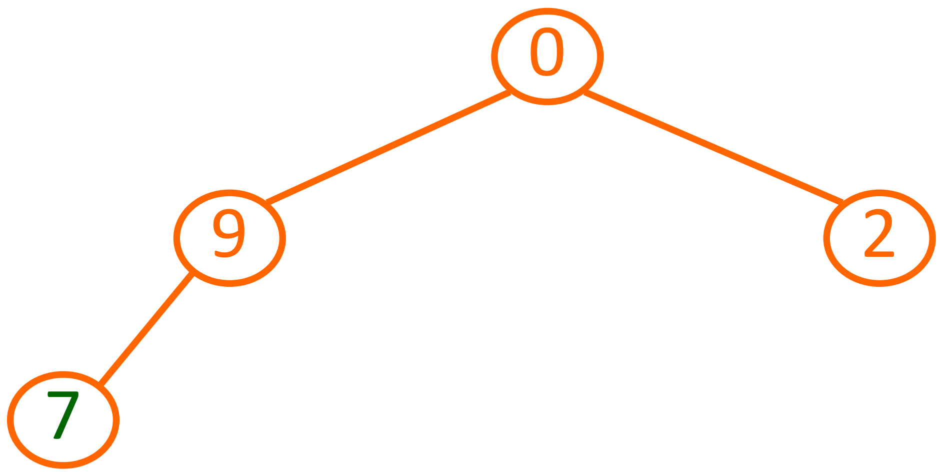
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	9	2	7						

Abstract Heap

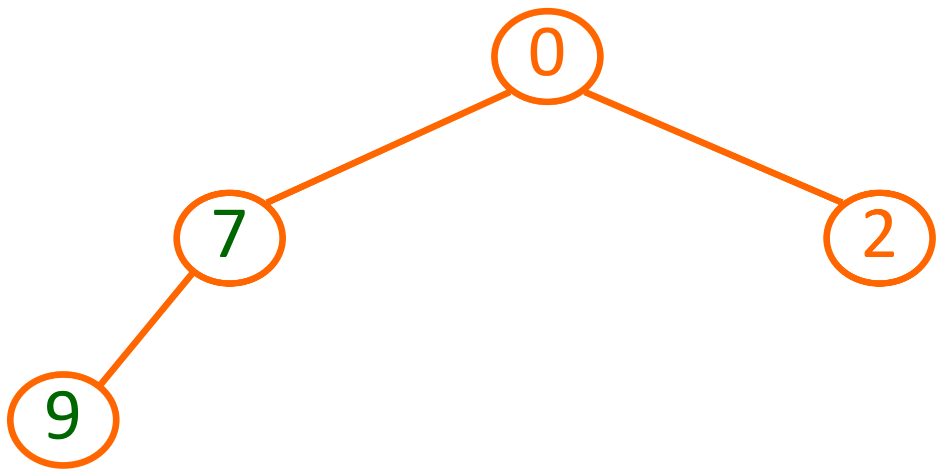


Murdoch UNIVERSITY

Heap Insert Animation

	2	9	0	7	8	6	5	1	4	3
Actual Heap	0	7	2	9						

Abstract Heap



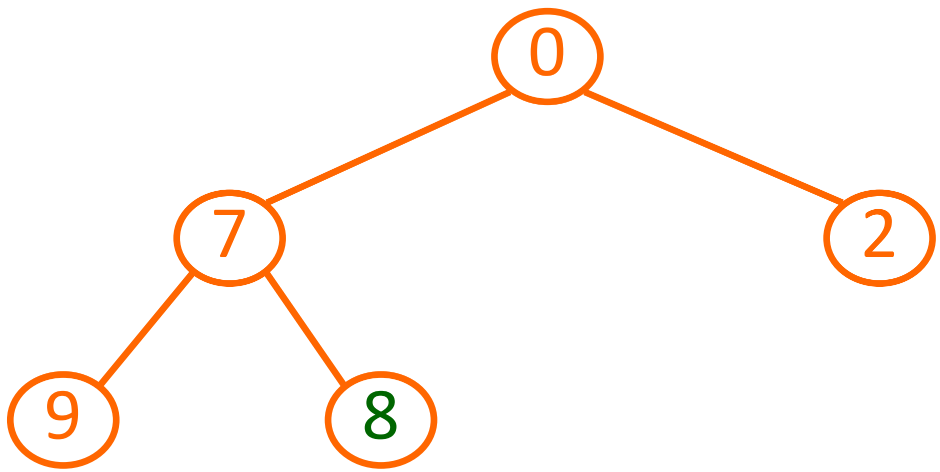
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	7	2	9	8					

Abstract Heap



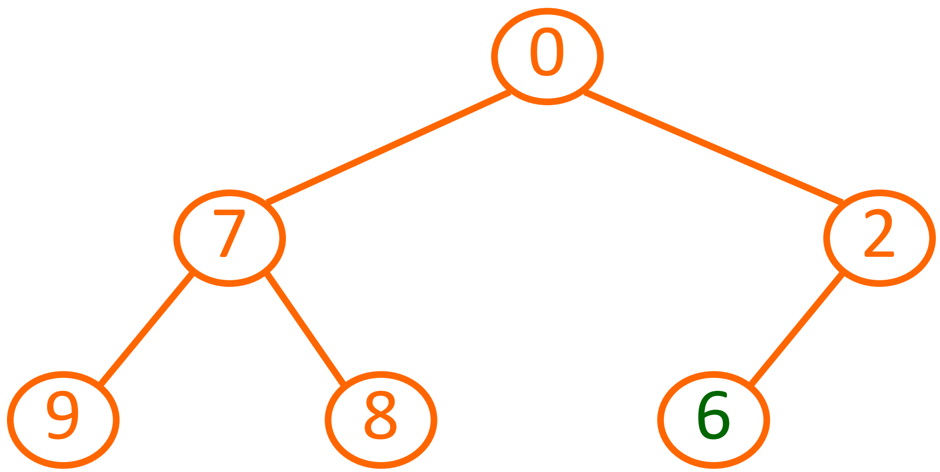
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	7	2	9	8	6				

Abstract Heap



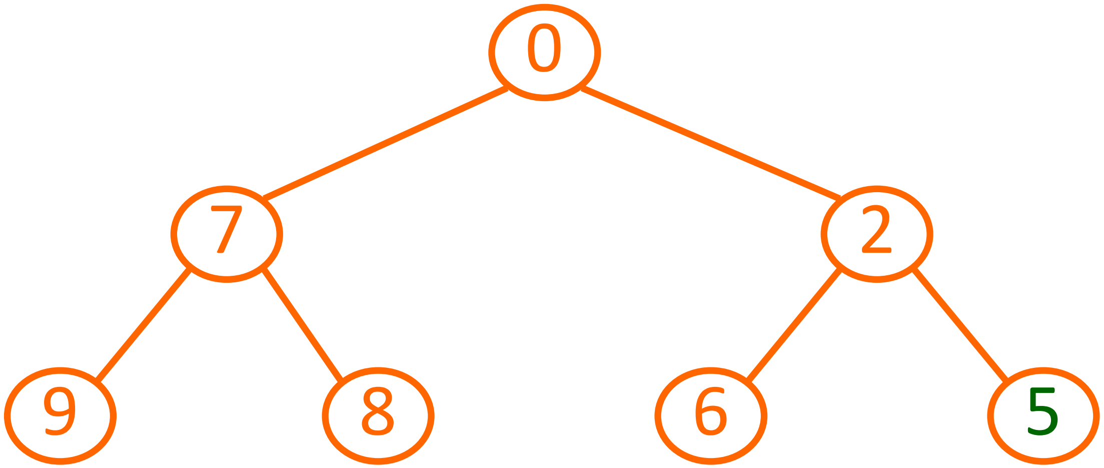
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	7	2	9	8	6	5			

Abstract Heap



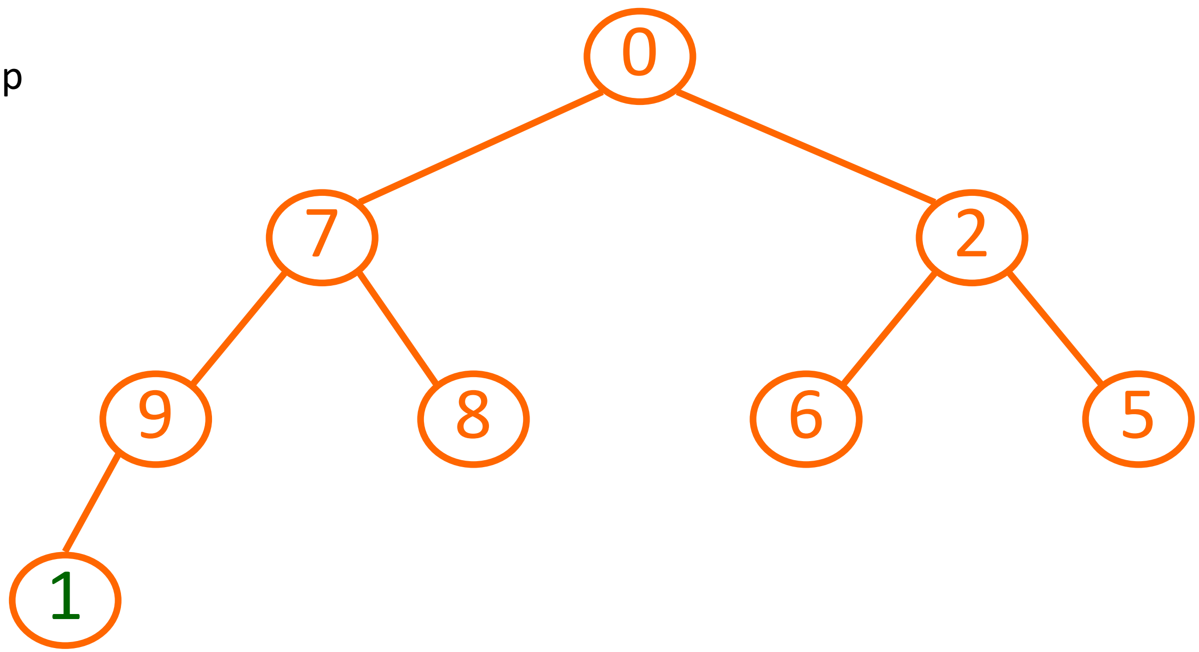
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	7	2	9	8	6	5	1		

Abstract Heap



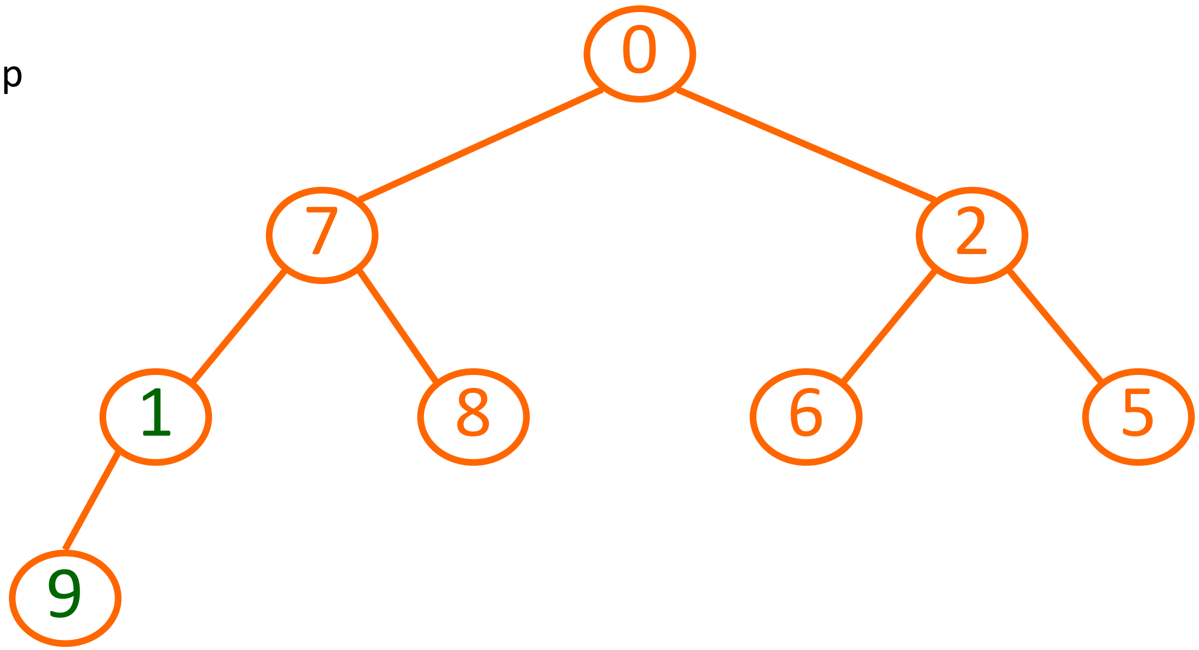
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	7	2	1	8	6	5	9		

Abstract Heap



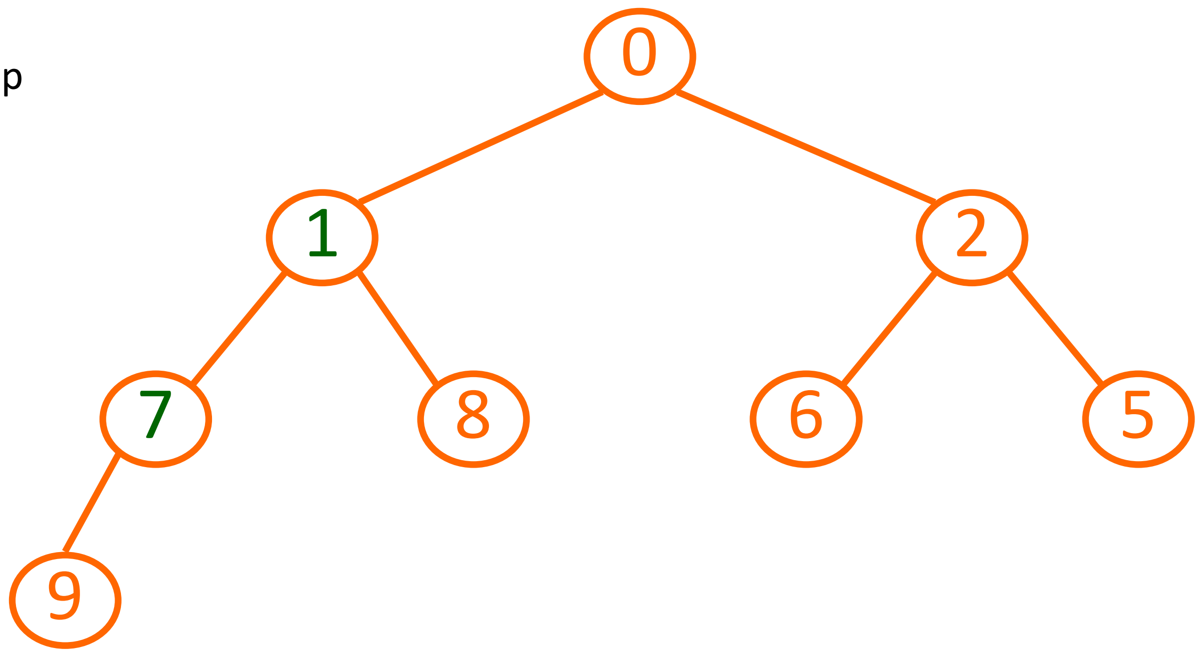
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	1	2	7	8	6	5	9		

Abstract Heap



Murdoch UNIVERSITY

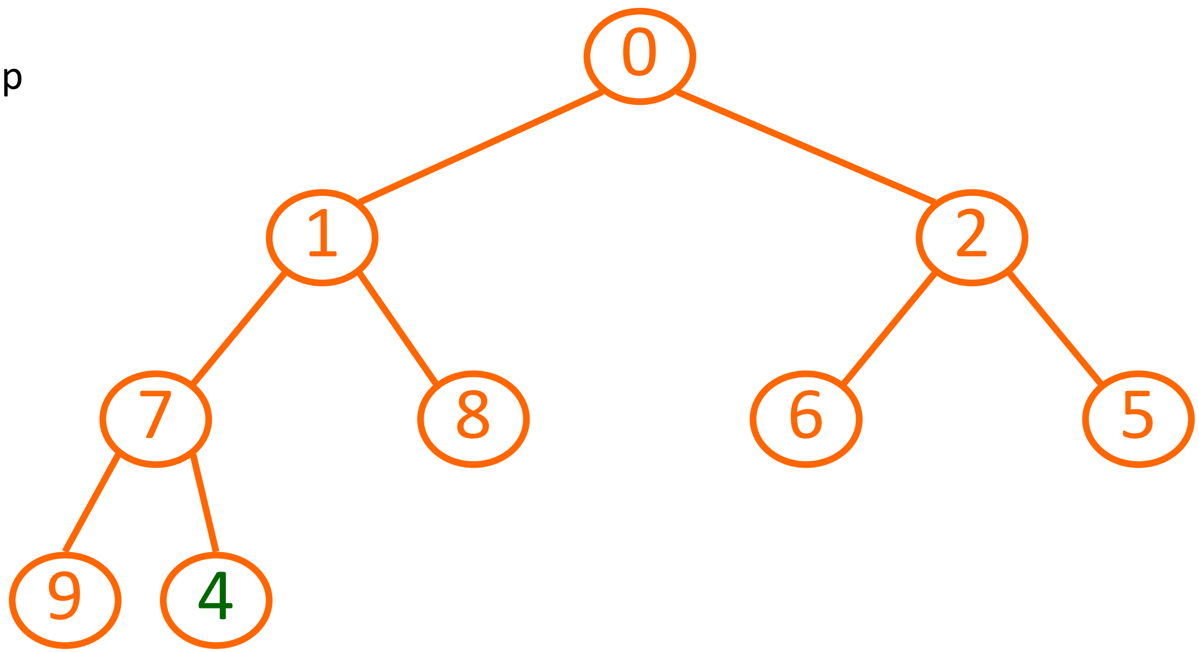
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	1	2	7	8	6	5	9	4	

Abstract Heap



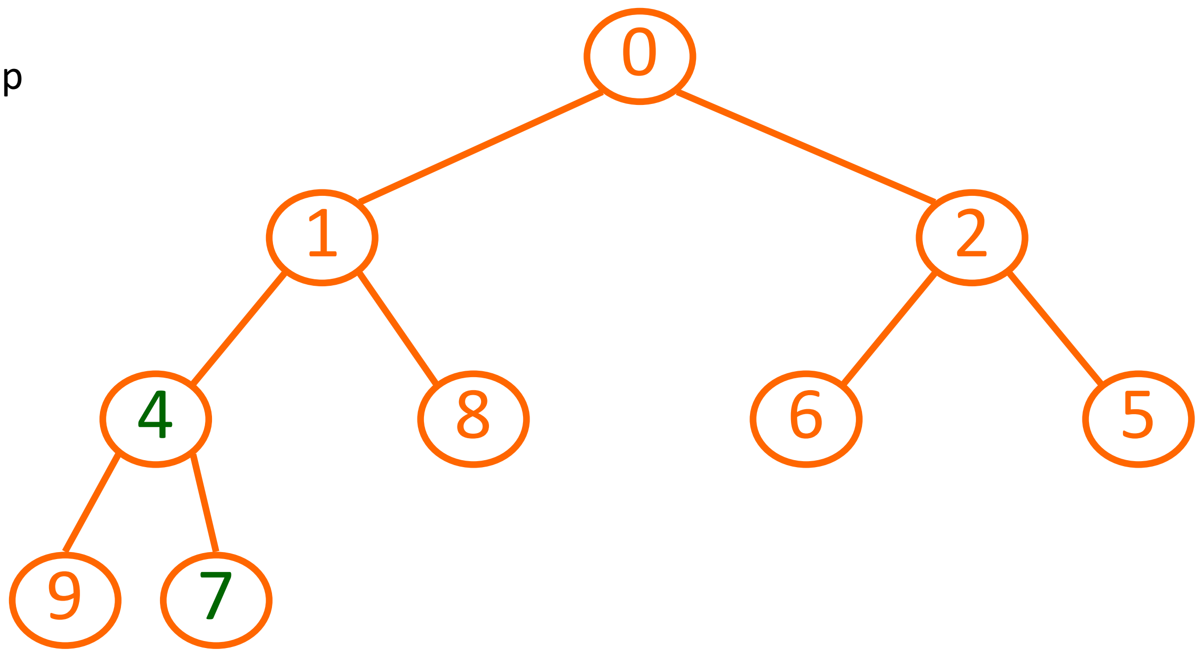
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	1	2	4	8	6	5	9	7	

Abstract Heap



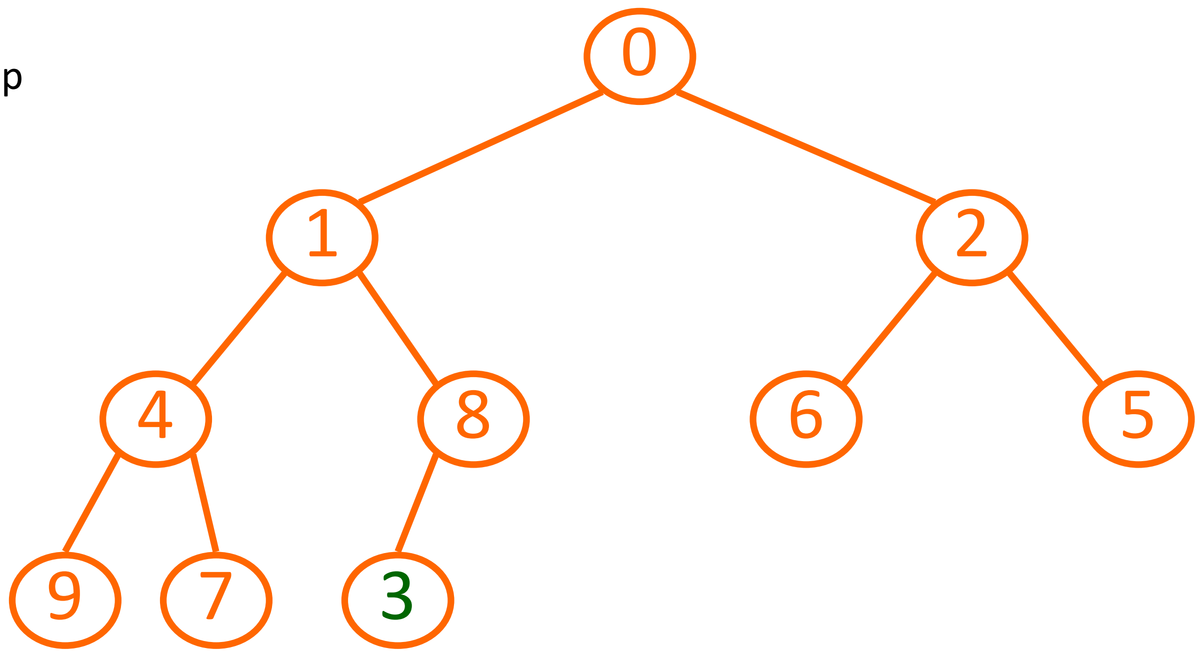
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	1	2	4	8	6	5	9	7	3

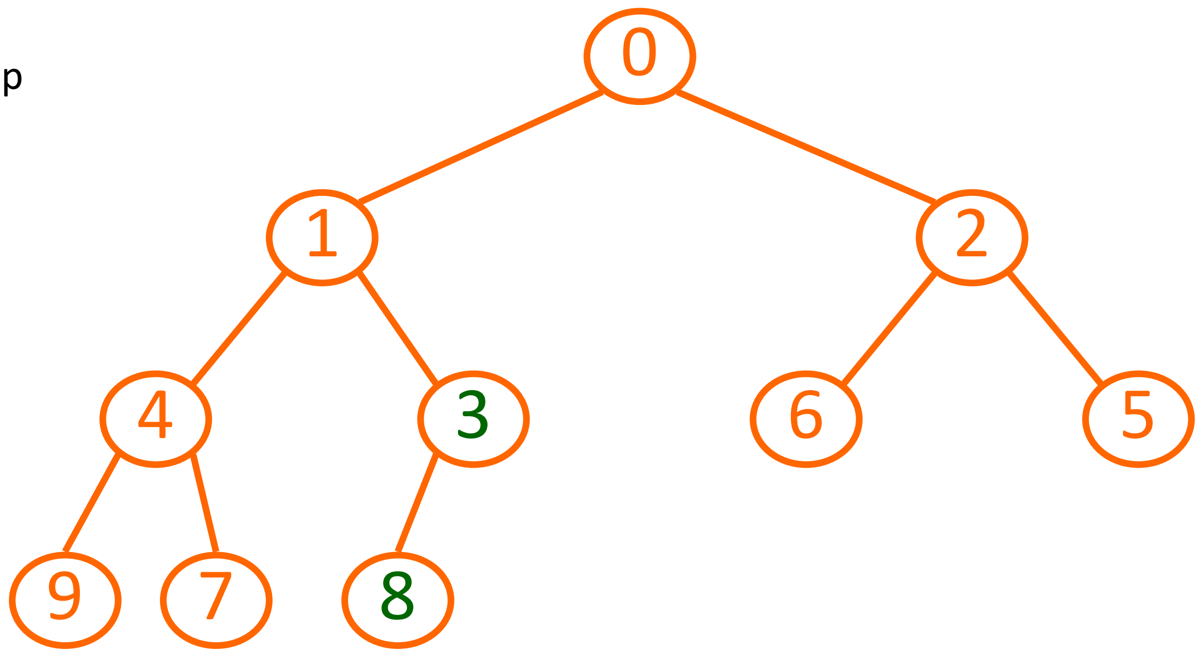
Abstract Heap



Heap Insert Animation

	2	9	0	7	8	6	5	1	4	3
Actual Heap	0	1	2	4	3	6	5	9	7	8

Abstract Heap



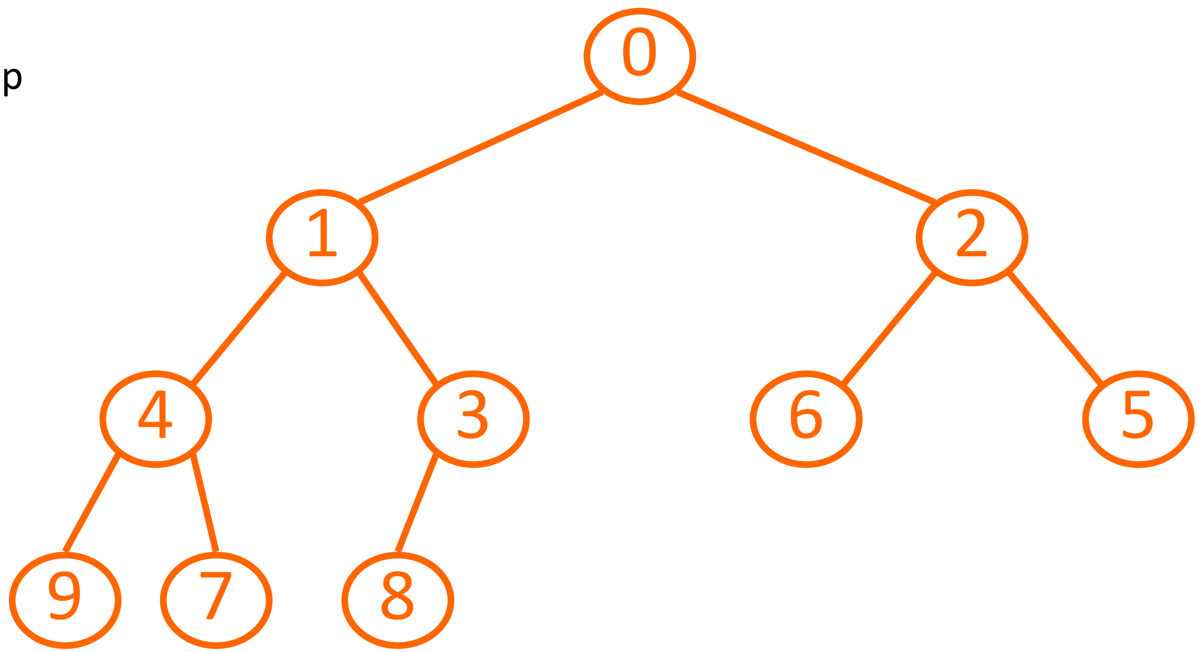
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	1	2	4	3	6	5	9	7	8

Abstract Heap



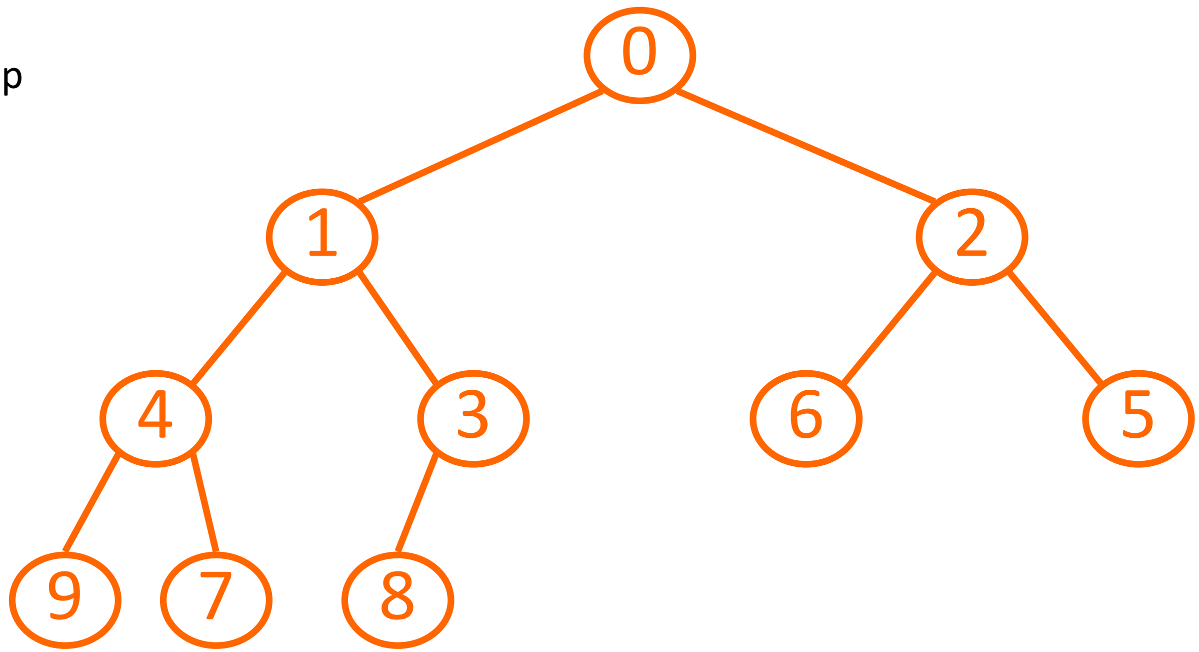
Heap Delete Animation

2	9	0	7	8	6	5	1	4	3

Actual Heap

0	1	2	4	3	6	5	9	7	8

Abstract Heap



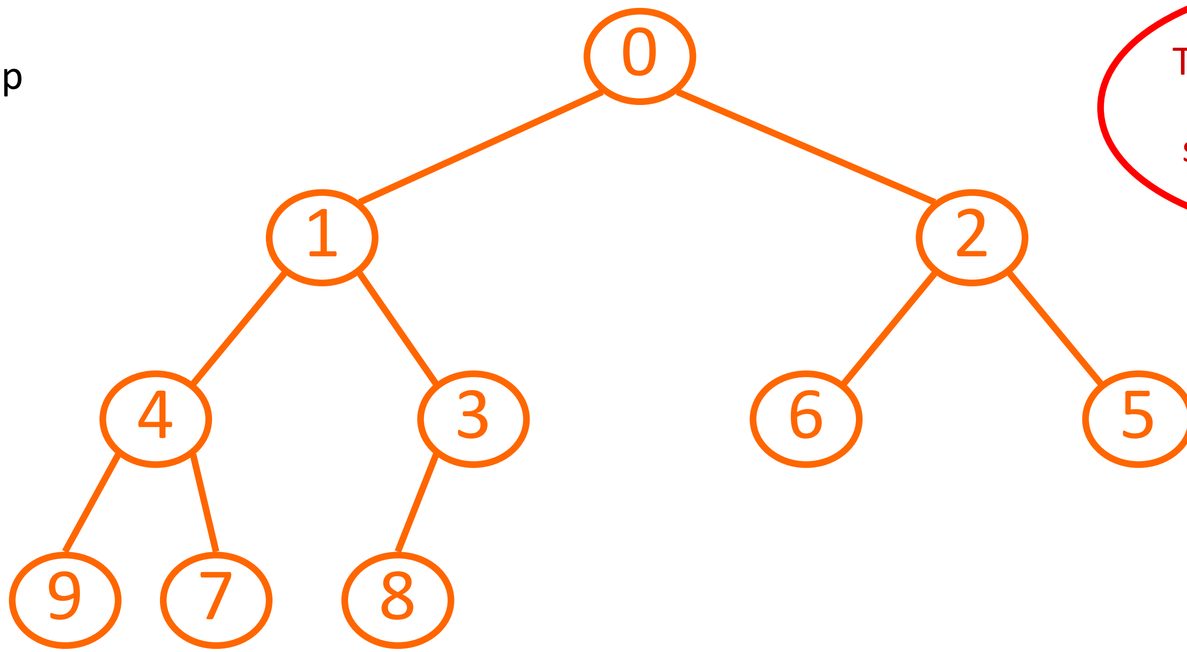
Heap Delete Animation

0	9	0	7	8	6	5	1	4	3

Actual Heap

0	1	2	4	3	6	5	9	7	8

Abstract Heap

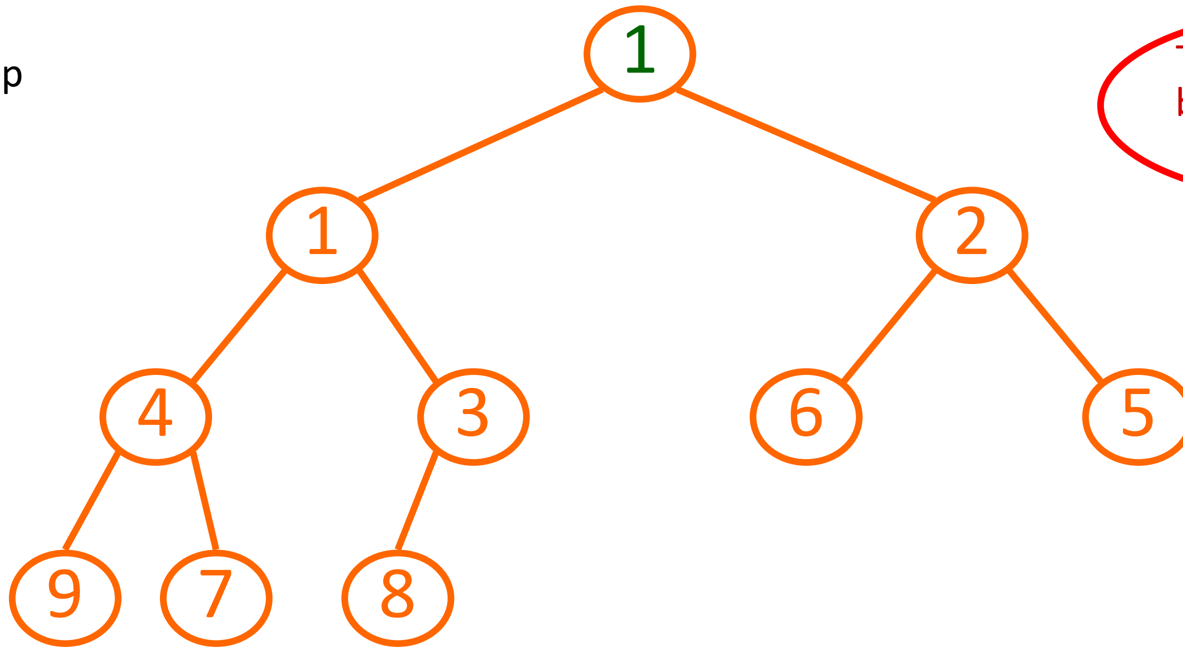


The top value goes onto the next space in the array

Heap Delete Animation

	0	9	0	7	8	6	5	1	4	3
Actual Heap	1	1	2	4	3	6	5	9	7	8

Abstract Heap

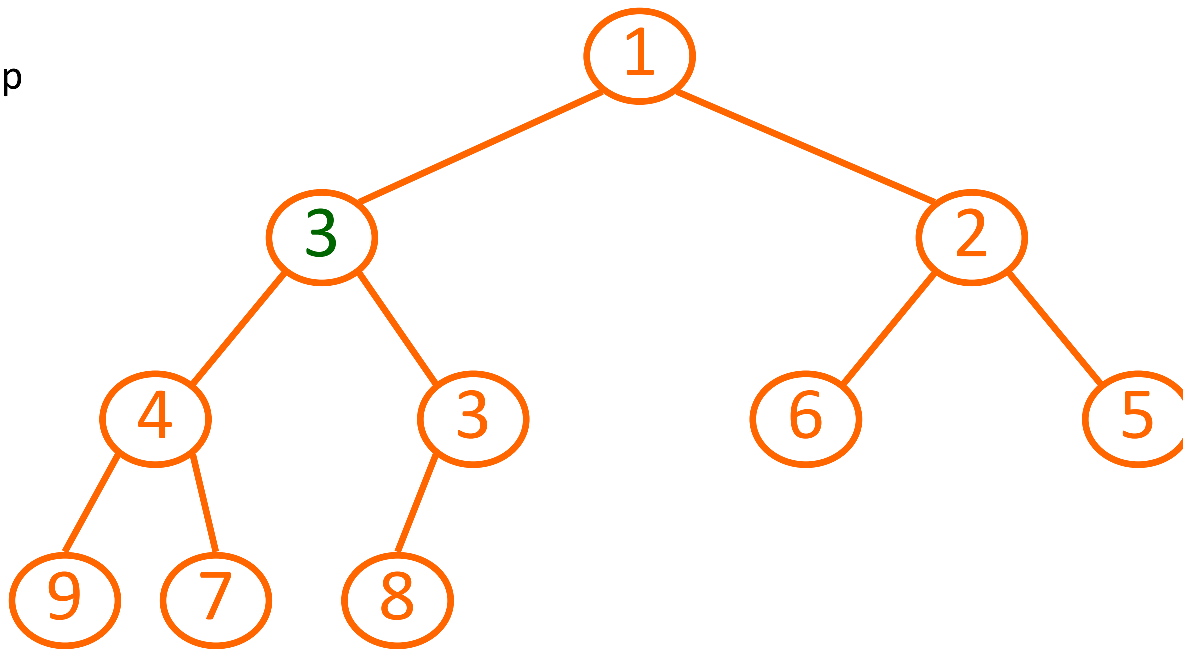


The smallest value below then moves upwards

Heap Delete Animation

	0	9	0	7	8	6	5	1	4	3
Actual Heap	1	3	2	4	3	6	5	9	7	8

Abstract Heap



And again

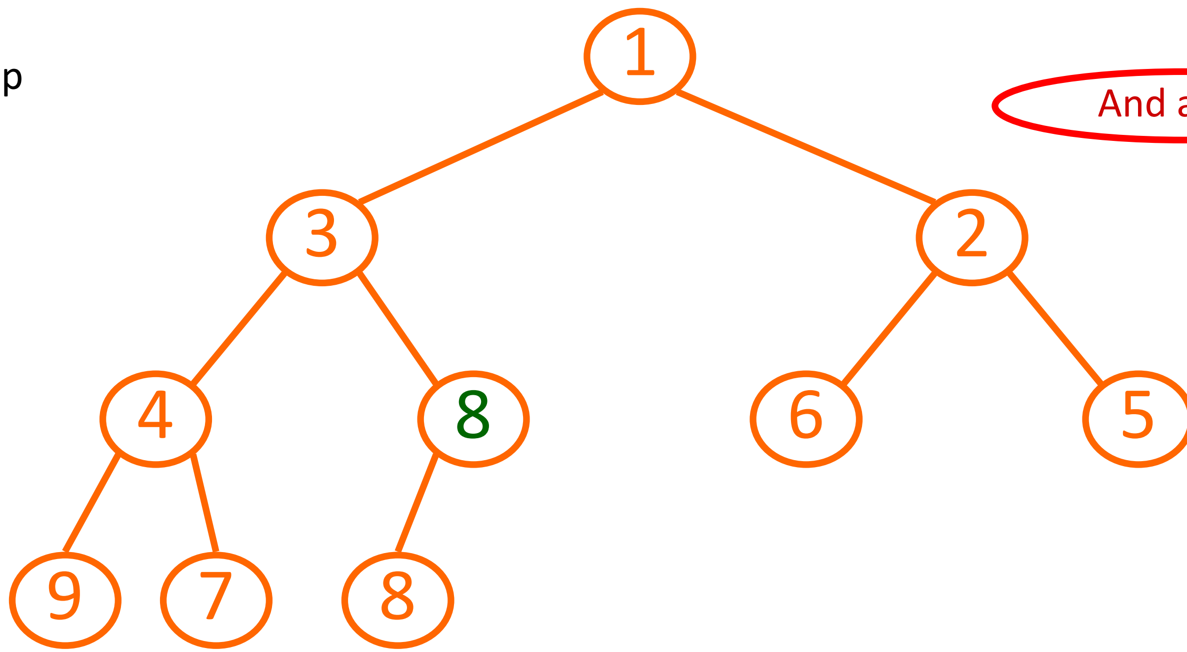
Heap Delete Animation

0	9	0	7	8	6	5	1	4	3

Actual Heap

1	3	2	4	8	6	5	9	7	8

Abstract Heap



And again

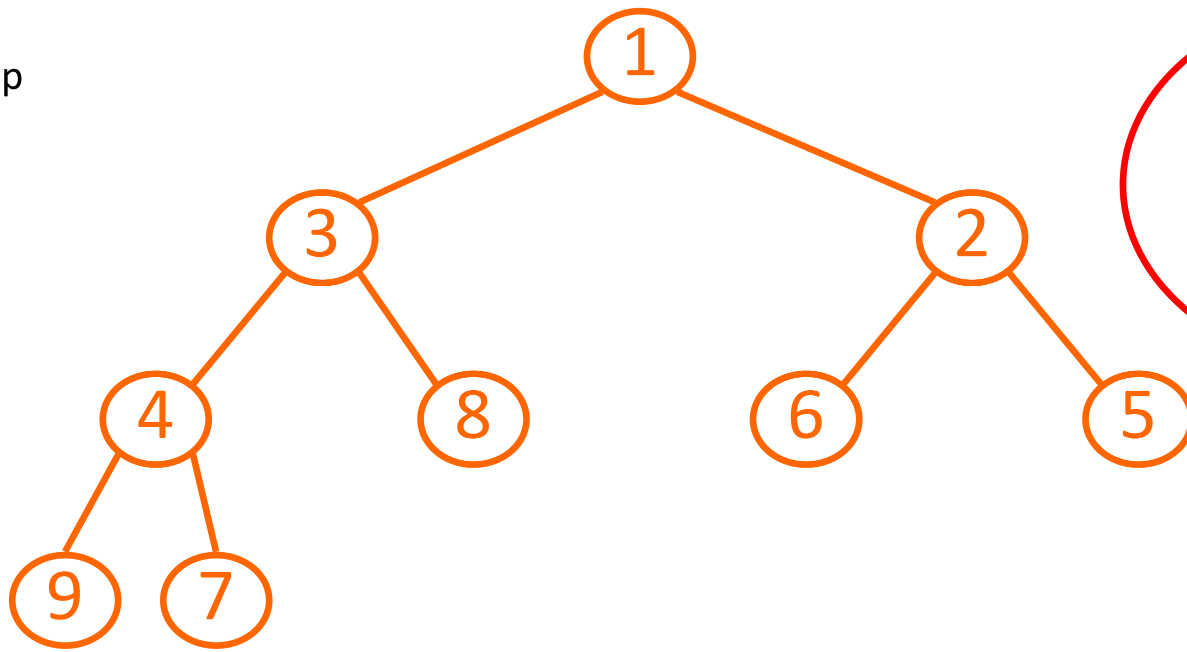
Heap Delete Animation

0	9	0	7	8	6	5	1	4	3

Actual Heap

1	3	2	4	8	6	5	9	7	

Abstract Heap



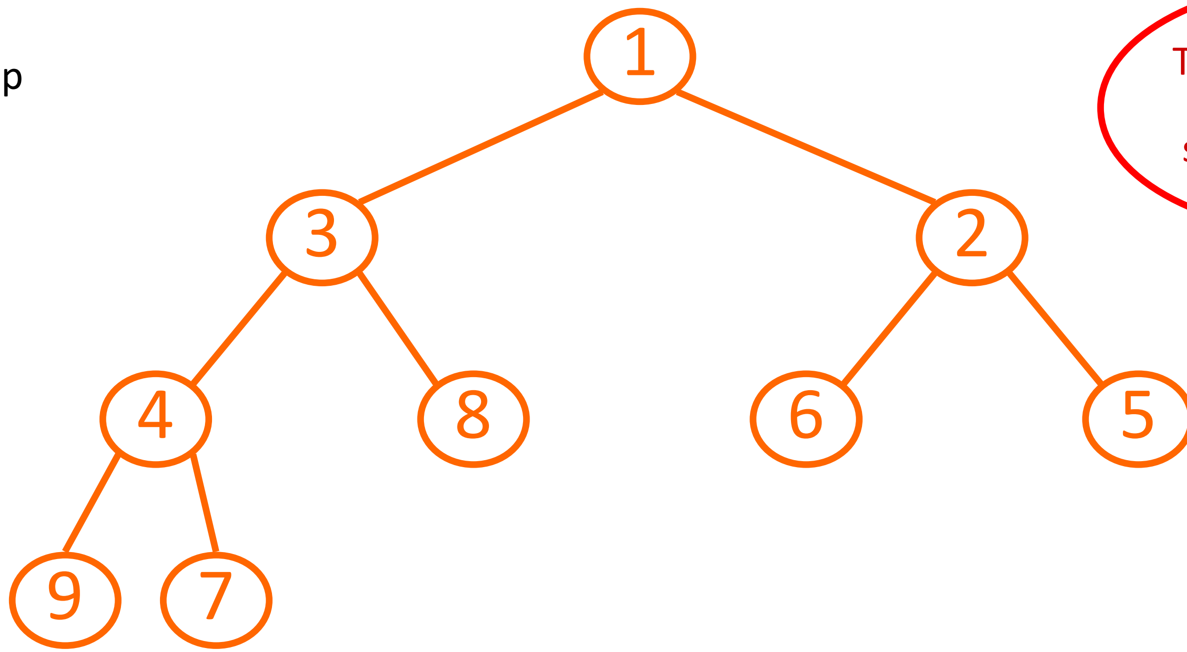
As we just moved the value in the right-most bottom node upwards, we simply 'delete' that node

Heap Delete Animation

0	1	0	7	8	6	5	1	4	3

Actual Heap	1	3	2	4	8	6	5	9	7	

Abstract Heap



The top value goes onto the next space in the array

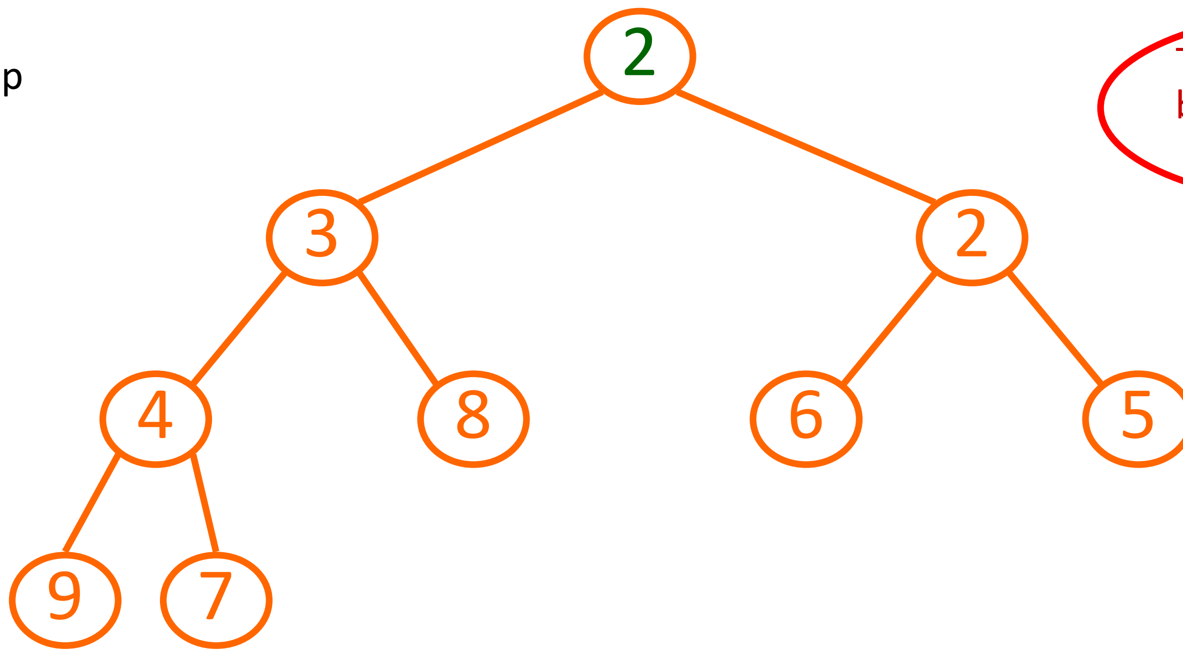
Heap Delete Animation

0	1	0	7	8	6	5	1	4	3

Actual Heap

2	3	2	4	8	6	5	9	7	

Abstract Heap



The smallest value below then moves upwards

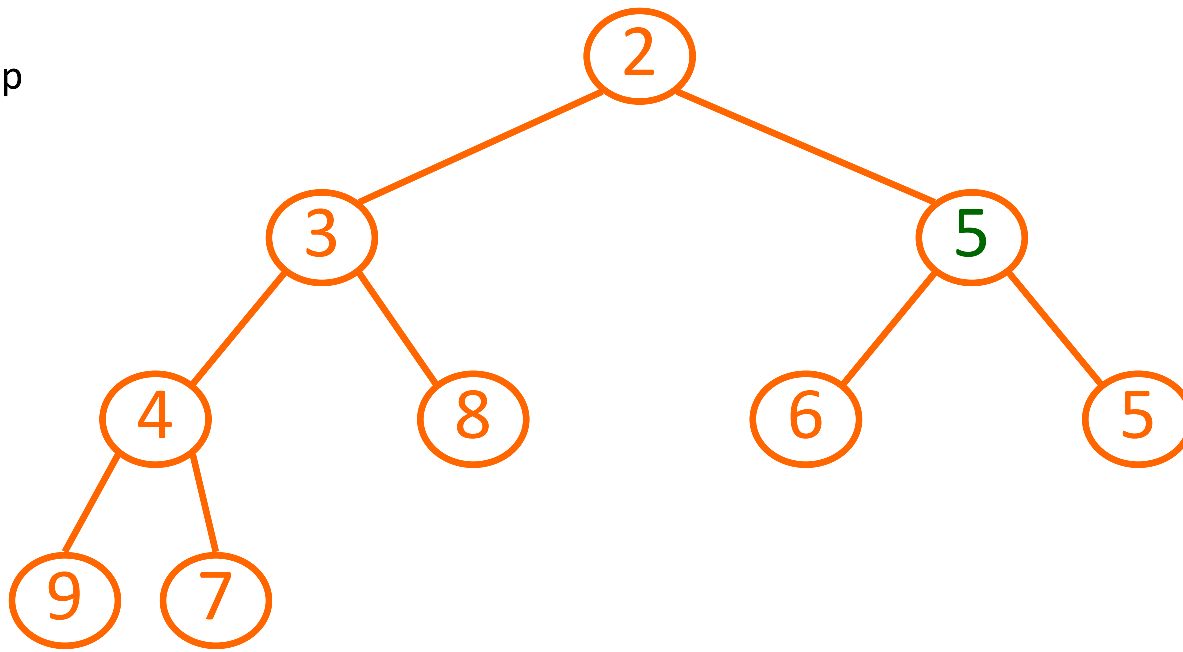
Heap Delete Animation

0	1	0	7	8	6	5	1	4	3

Actual Heap

2	3	5	4	8	6	5	9	7	

Abstract Heap

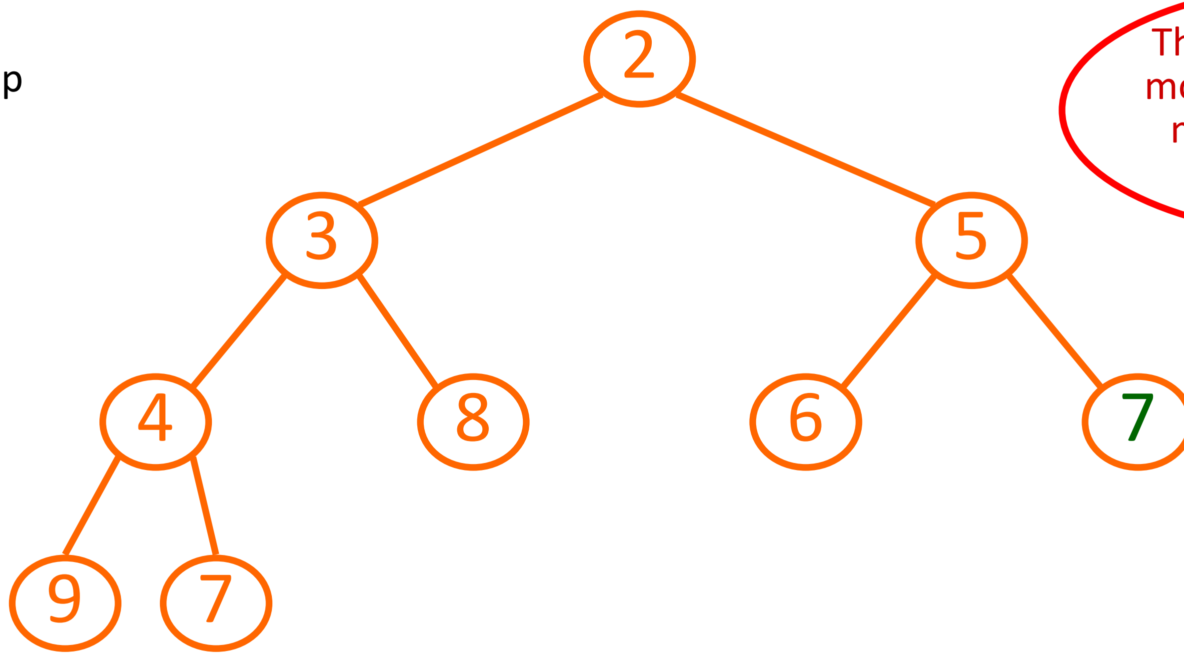


And again

Heap Delete Animation

	0	1	0	7	8	6	5	1	4	3
Actual Heap	2	3	5	4	8	6	7	9	7	

Abstract Heap



The value in the right-most bottom row, now moves into this free node

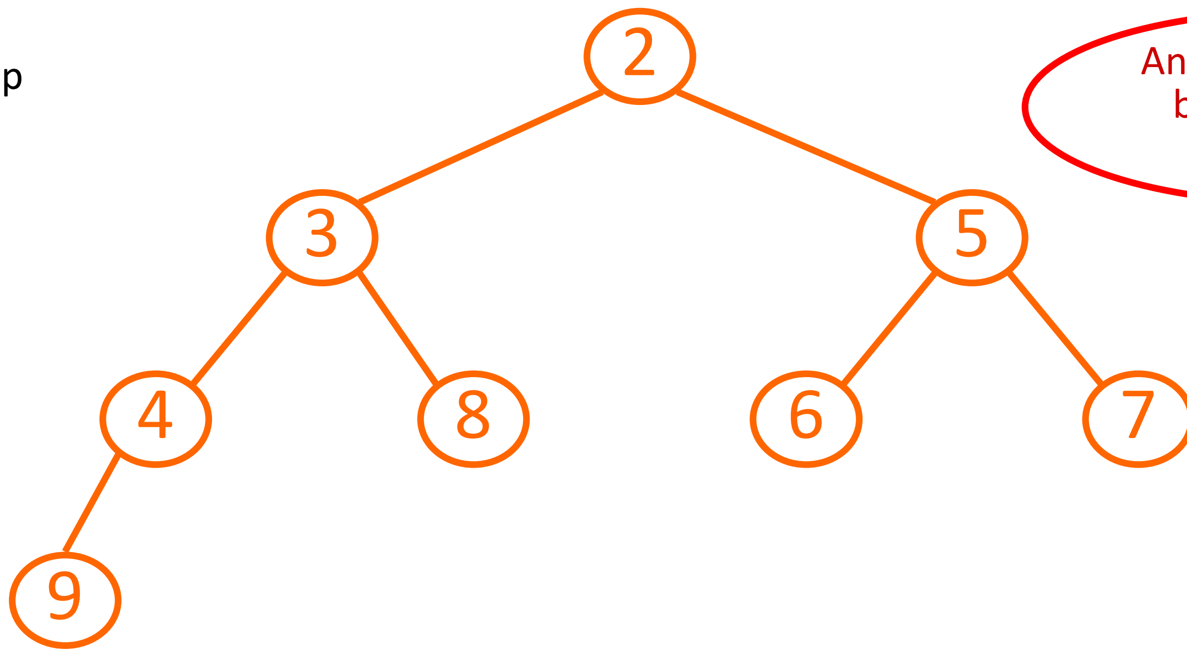
Heap Delete Animation

0	1	0	7	8	6	5	1	4	3

Actual Heap

2	3	5	4	8	6	7	9		

Abstract Heap



And the right-most bottom node is 'deleted'

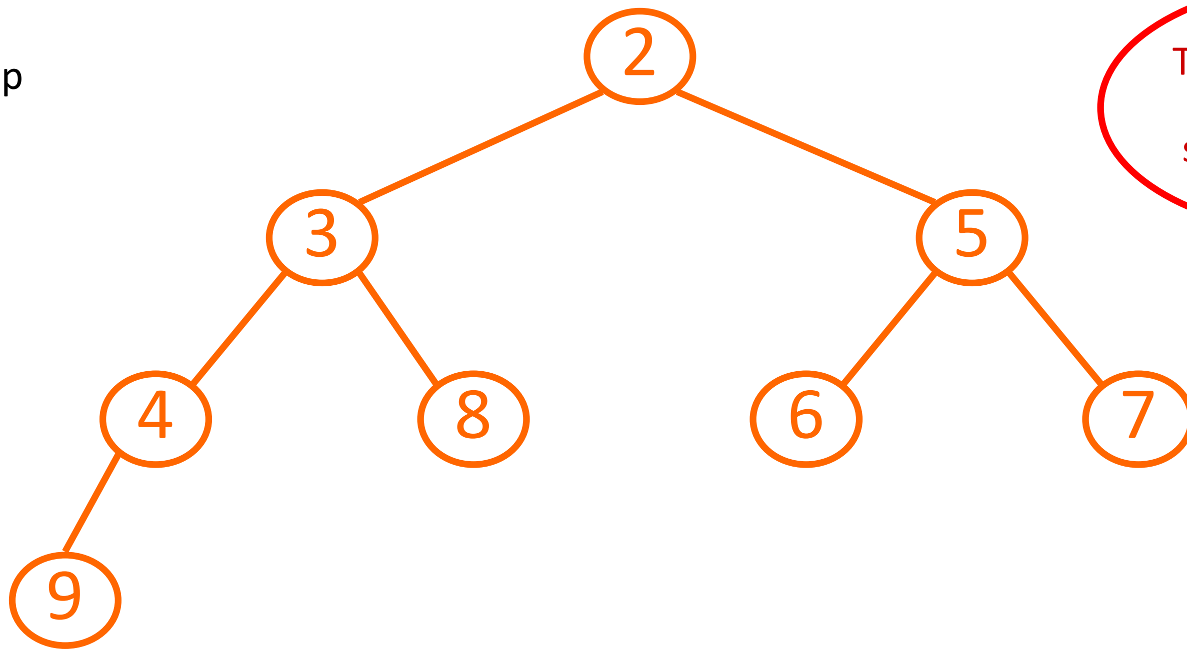
Heap Delete Animation

0	1	2	7	8	6	5	1	4	3

Actual Heap

2	3	5	4	8	6	7	9		

Abstract Heap

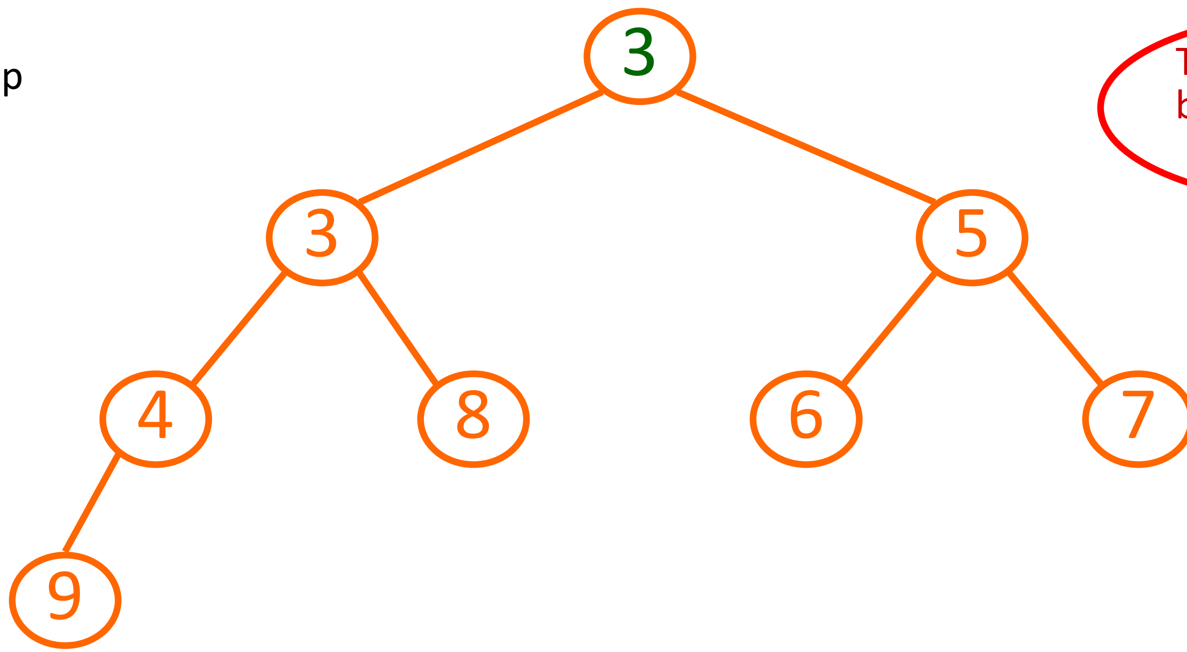


The top value goes onto the next space in the array

Heap Delete Animation

	0	1	2	7	8	6	5	1	4	3
Actual Heap	3	3	5	4	8	6	7	9		

Abstract Heap



The smallest value below then moves upwards

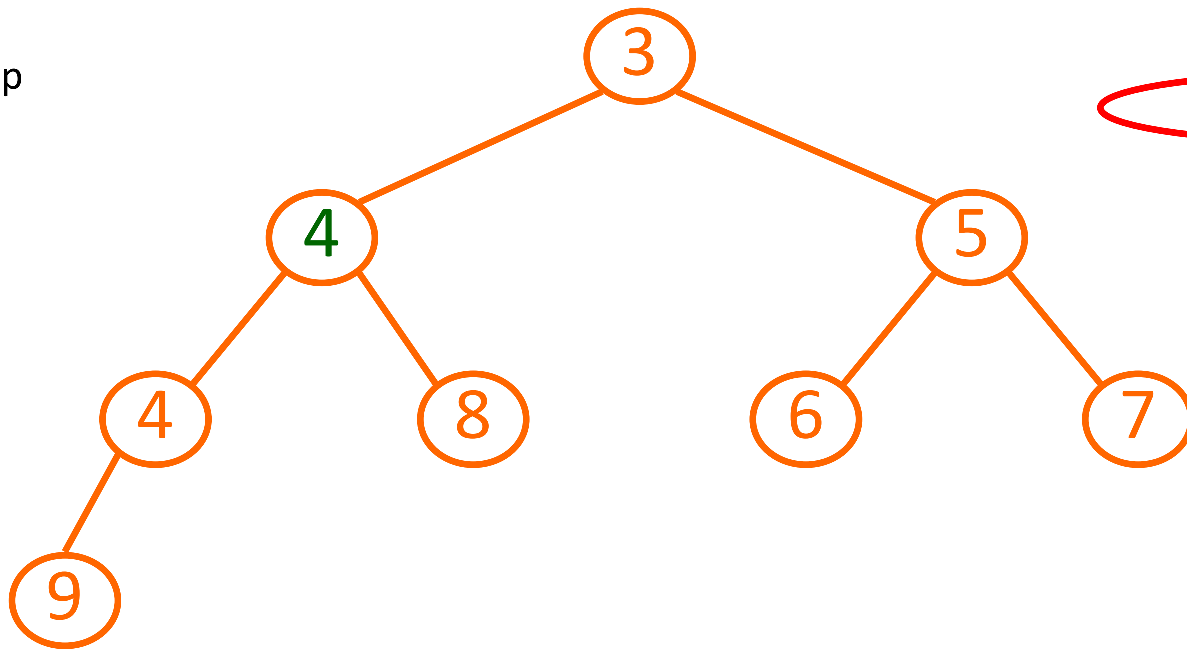
Heap Delete Animation

0	1	2	7	8	6	5	1	4	3

Actual Heap

3	4	5	4	8	6	7	9		

Abstract Heap



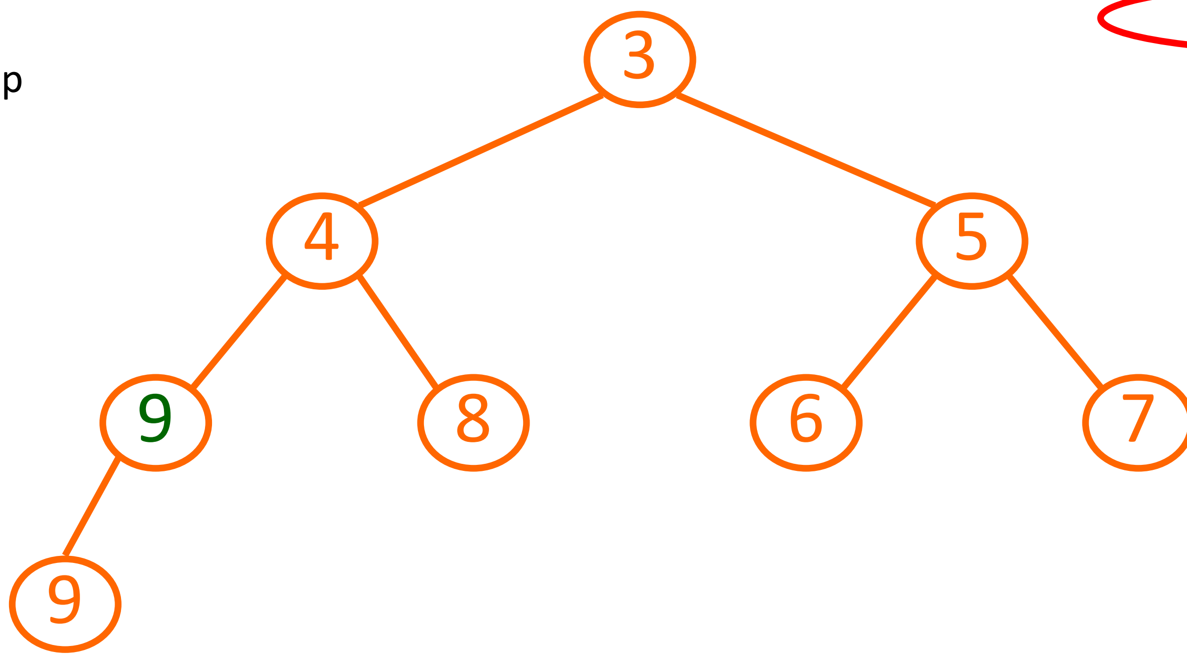
And again

Heap Delete Animation

	0	1	2	7	8	6	5	1	4	3
Actual Heap	3	4	5	9	8	6	7	9		

And again

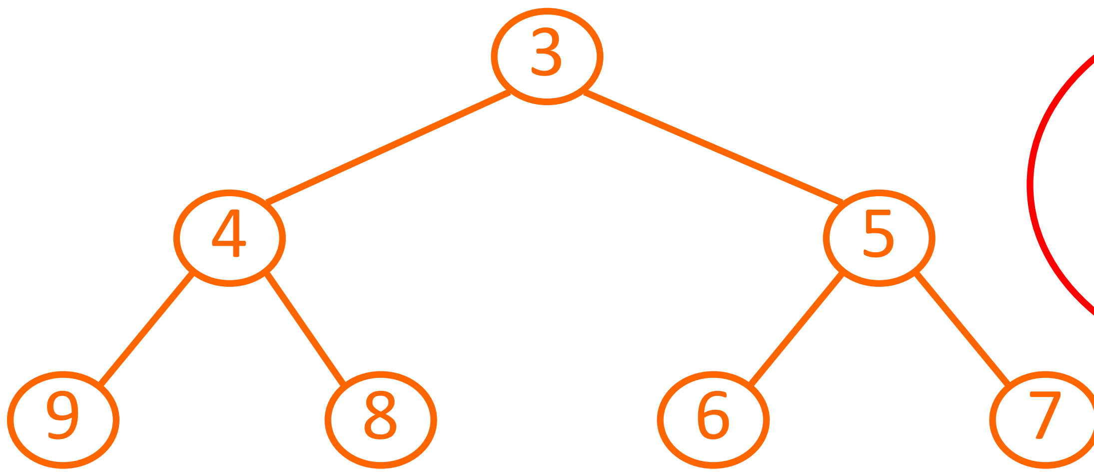
Abstract Heap



Heap Delete Animation

	0	1	2	7	8	6	5	1	4	3
Actual Heap	3	4	5	9	8	6	7			

Abstract Heap



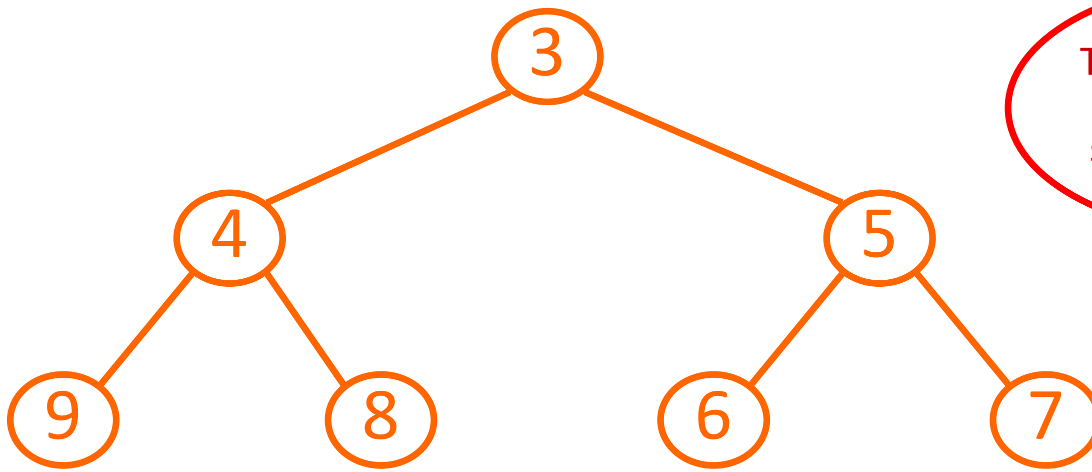
As we just moved the value in the right-most bottom node upwards, we simply 'delete' that node

Heap Delete Animation

0	1	2	3	8	6	5	1	4	3

Actual Heap										
	3	4	5	9	8	6	7			

Abstract Heap



The top value goes onto the next space in the array

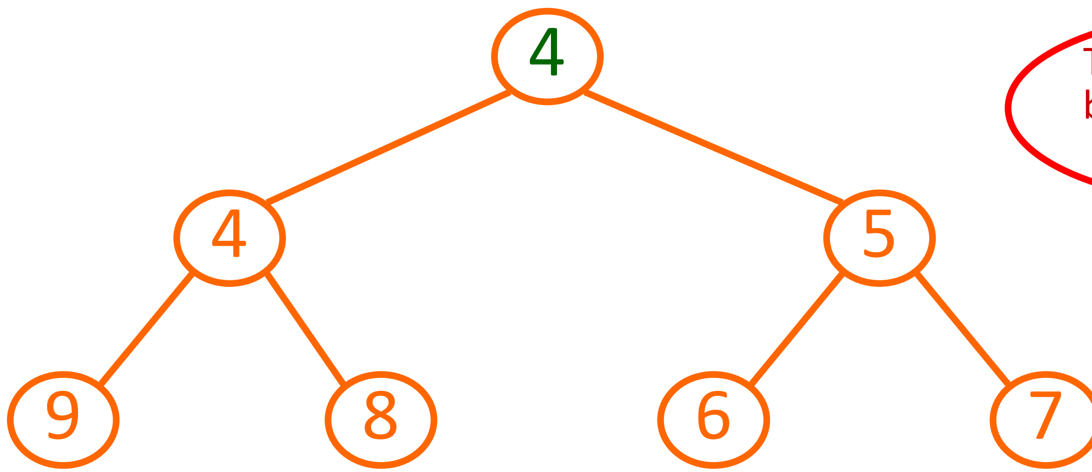
Heap Delete Animation

0	1	2	3	8	6	5	1	4	3

Actual Heap

4	4	5	9	8	6	7			

Abstract Heap



The smallest value below then moves upwards

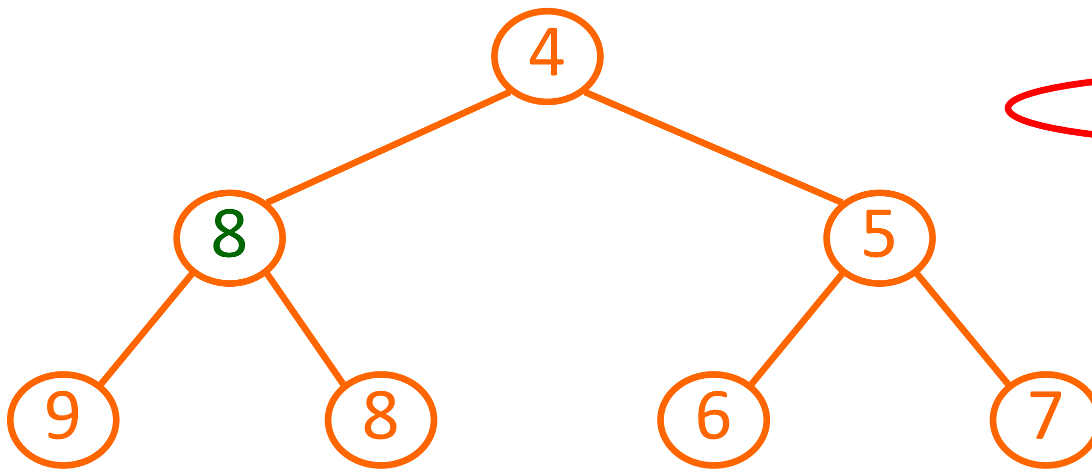
Heap Delete Animation

0	1	2	3	8	6	5	1	4	3

Actual Heap

4	8	5	9	8	6	7			

Abstract Heap

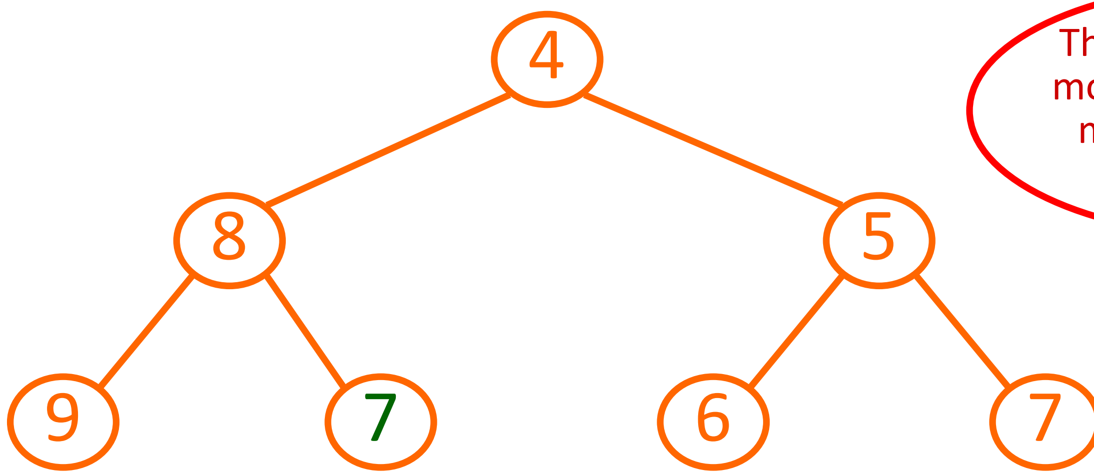


And again

Heap Delete Animation

	0	1	2	3	8	6	5	1	4	3
Actual Heap	4	8	5	9	7	6	7			

Abstract Heap

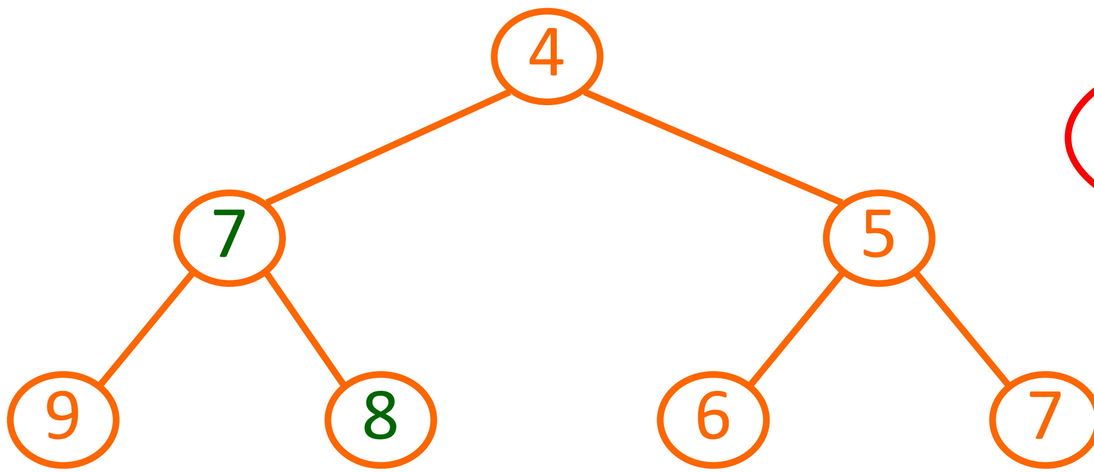


The value in the right-most bottom row, now moves into this free node

Heap Delete Animation

	0	1	2	3	8	6	5	1	4	3
Actual Heap	4	7	5	9	8	6	7			

Abstract Heap

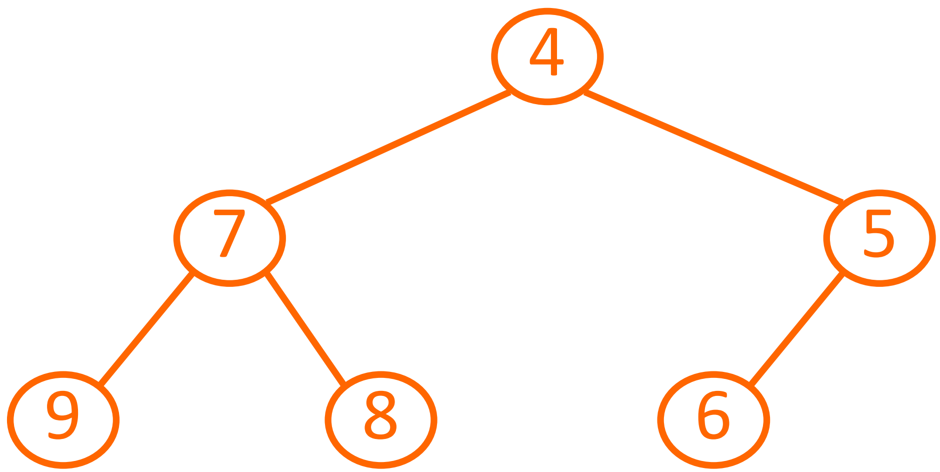


And 'percolates' upwards

Heap Delete Animation

	0	1	2	3	8	6	5	1	4	3
Actual Heap	4	7	5	9	8	6				

Abstract Heap



And the right-most bottom node is deleted

Murdoch UNIVERSITY

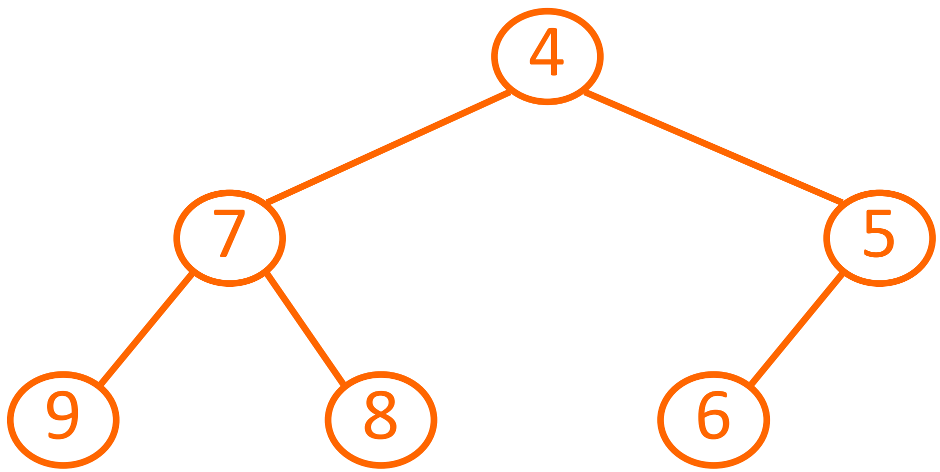
Heap Delete Animation

0	1	2	3	4	6	5	1	4	3

Actual Heap

4	7	5	9	8	6				

Abstract Heap

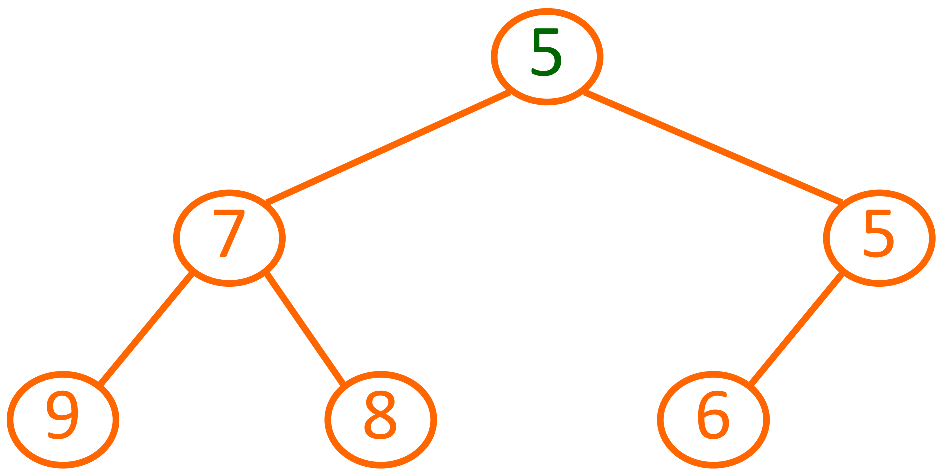


Murdoch UNIVERSITY

Heap Delete Animation

	0	1	2	3	4	6	5	1	4	3
Actual Heap	5	7	5	9	8	6				

Abstract Heap



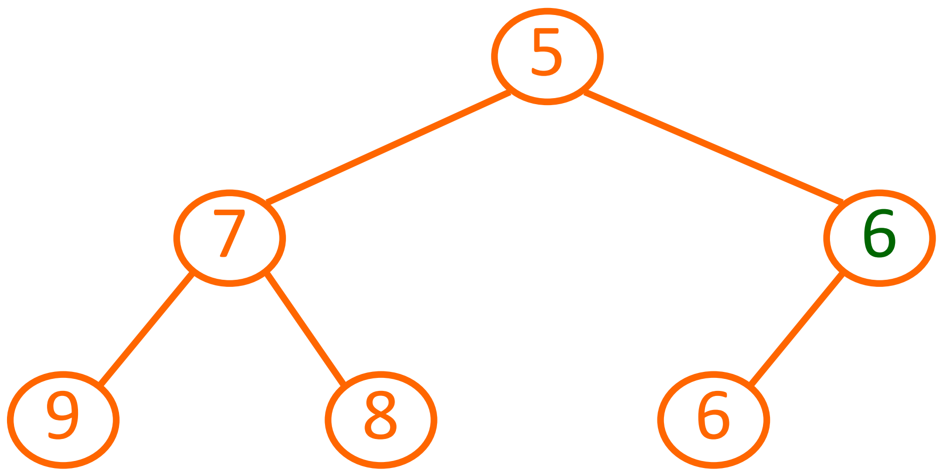
Heap Delete Animation

0	1	2	3	4	6	5	1	4	3

Actual Heap

5	7	6	9	8	6				

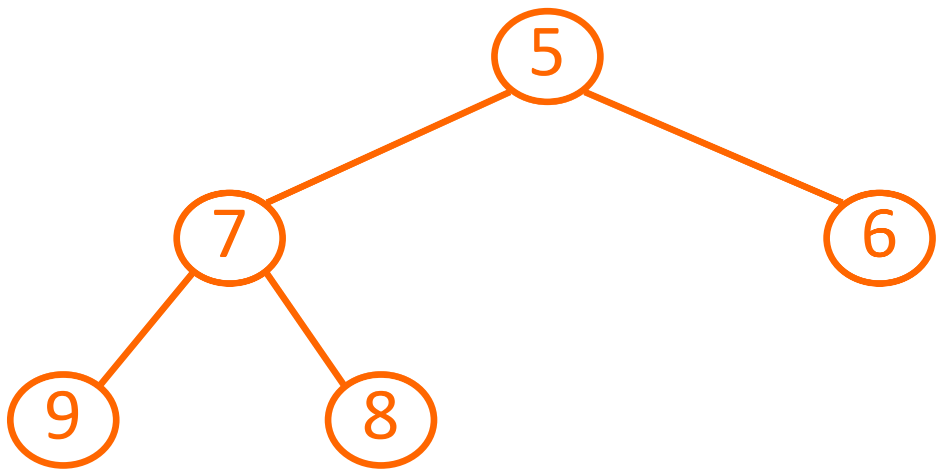
Abstract Heap



Heap Delete Animation

	0	1	2	3	4	6	5	1	4	3
Actual Heap	5	7	6	9	8					

Abstract Heap



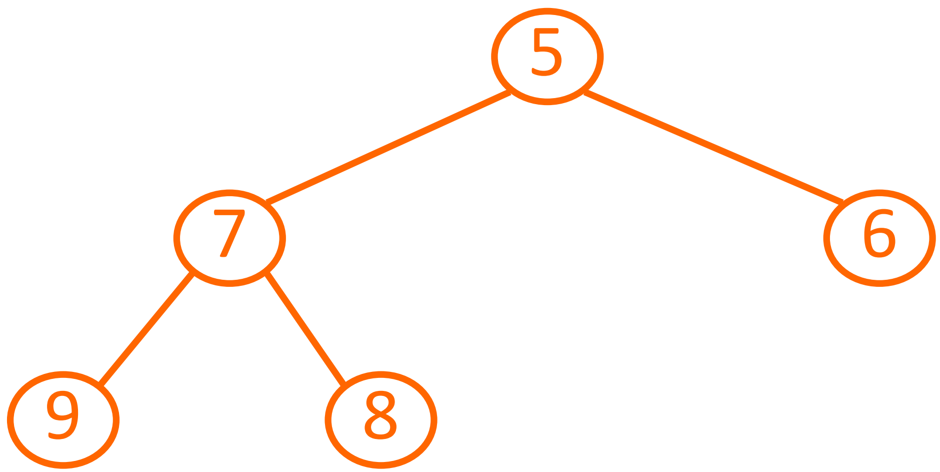
Heap Delete Animation

0	1	2	3	4	5	5	1	4	3

Actual Heap

5	7	6	9	8					

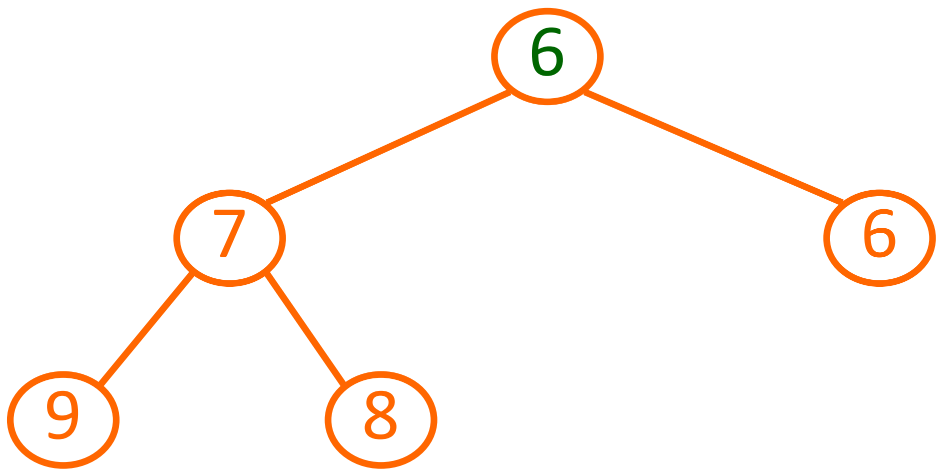
Abstract Heap



Heap Delete Animation

	0	1	2	3	4	5	5	1	4	3
Actual Heap	6	7	6	9	8					

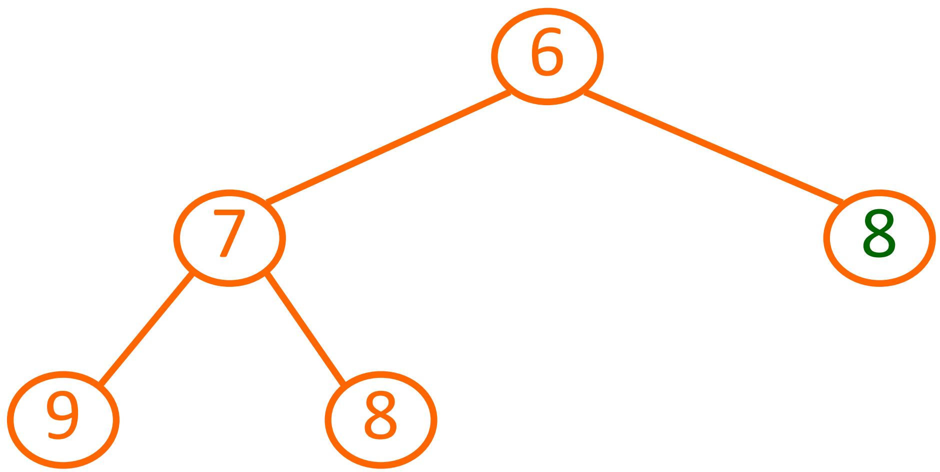
Abstract Heap



Heap Delete Animation

	0	1	2	3	4	5	5	1	4	3
Actual Heap	6	7	8	9	8					

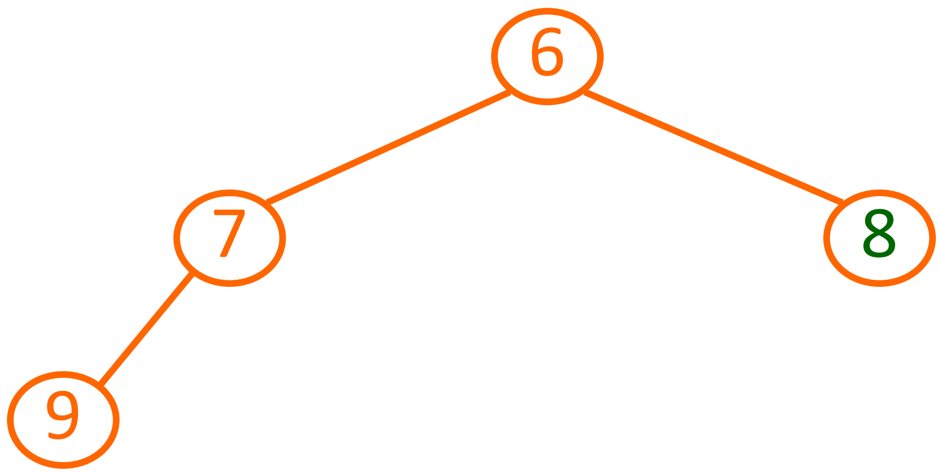
Abstract Heap



Heap Delete Animation

	0	1	2	3	4	5	5	1	4	3
Actual Heap	6	7	8	9						

Abstract Heap



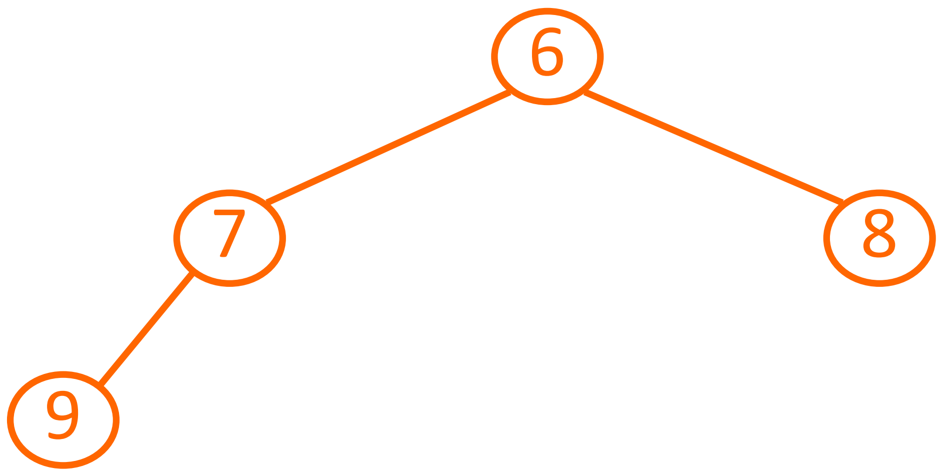
Heap Delete Animation

0	1	2	3	4	5	6	1	4	3

Actual Heap

6	7	8	9						

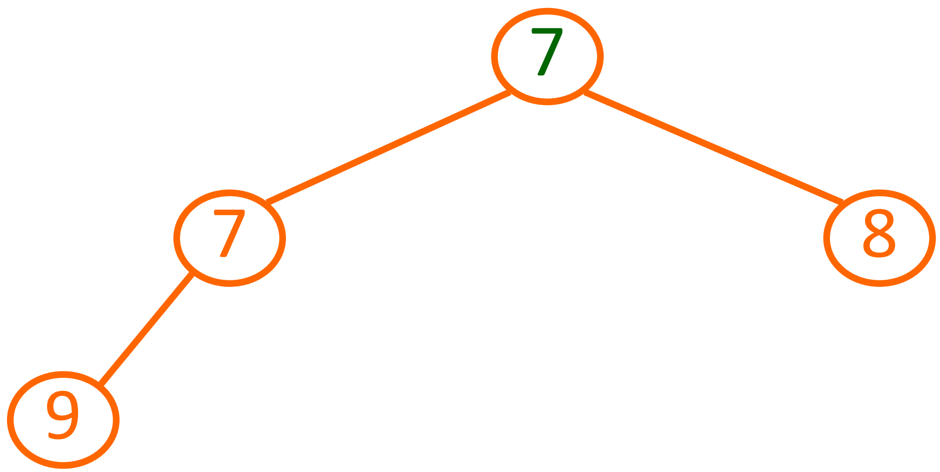
Abstract Heap



Heap Delete Animation

	0	1	2	3	4	5	6	1	4	3
Actual Heap	7	7	8	9						

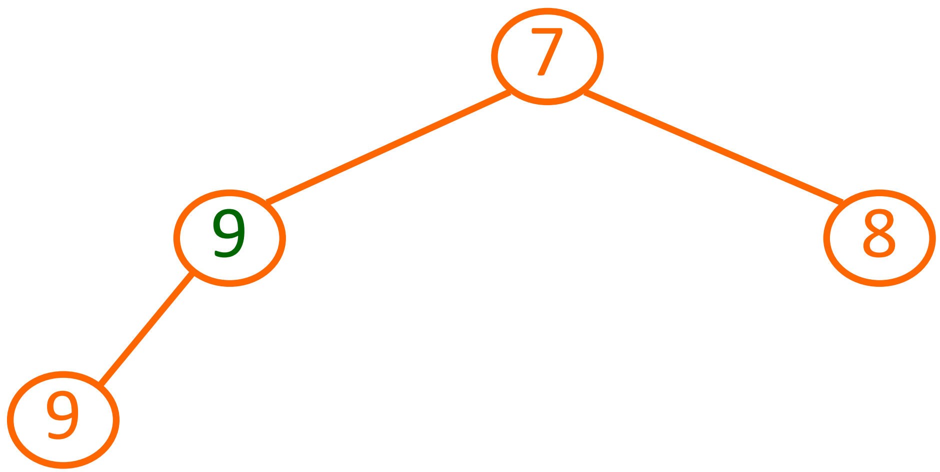
Abstract Heap



Heap Delete Animation

	0	1	2	3	4	5	6	1	4	3
Actual Heap	7	9	8	9						

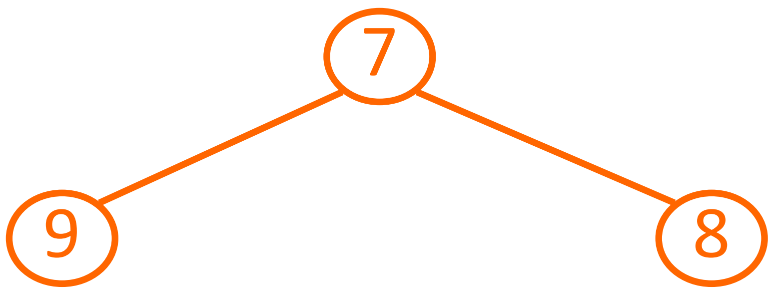
Abstract Heap



Heap Delete Animation

	0	1	2	3	4	5	6	1	4	3
Actual Heap	7	9	8							

Abstract Heap



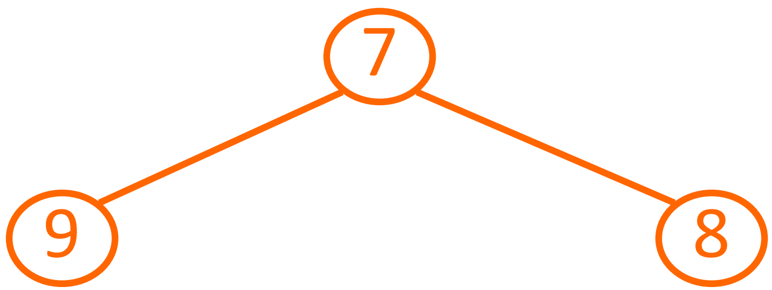
Heap Delete Animation

0	1	2	3	4	5	6	7	4	3

Actual Heap

7	9	8							

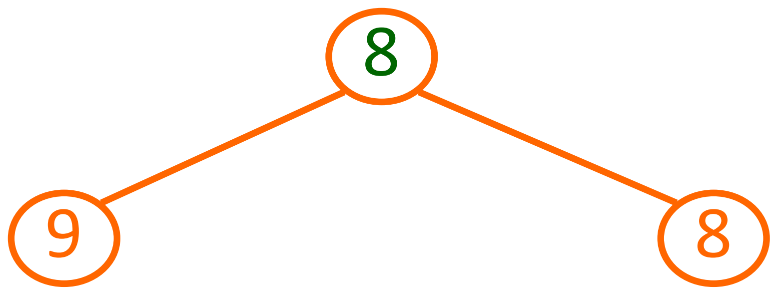
Abstract Heap



Heap Delete Animation

	0	1	2	3	4	5	6	7	4	3
Actual Heap	8	9	8							

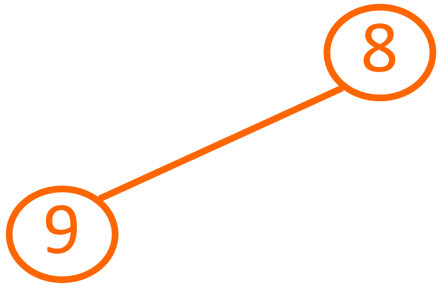
Abstract Heap



Heap Delete Animation

	0	1	2	3	4	5	6	7	4	3
Actual Heap	8	9								

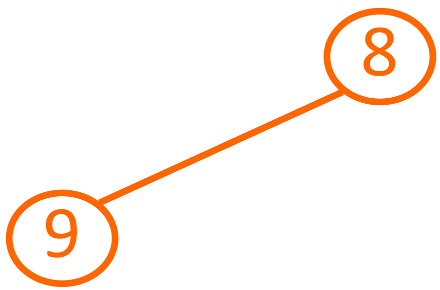
Abstract Heap



Heap Delete Animation

	0	1	2	3	4	5	6	7	8	3
Actual Heap	8	9								

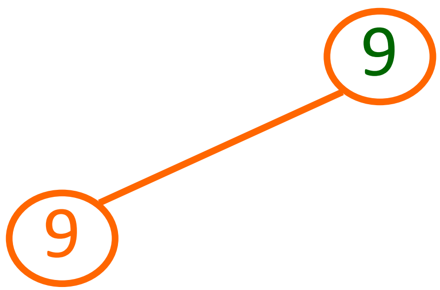
Abstract Heap



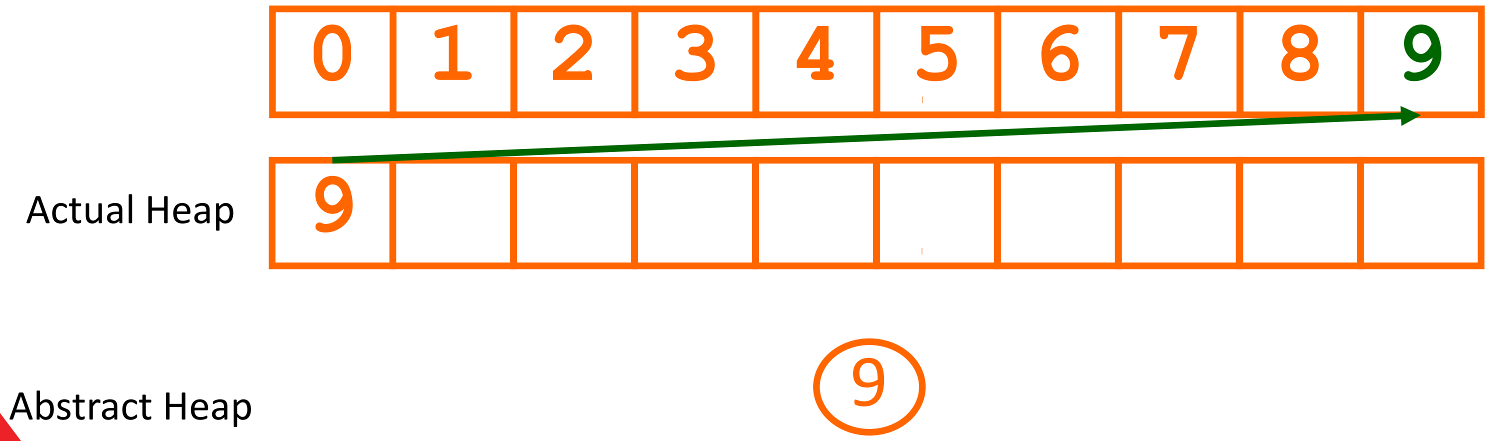
Heap Delete Animation

	0	1	2	3	4	5	6	7	8	3
Actual Heap	9	9								

Abstract Heap



Heap Delete Animation



Heap Delete Animation

	0	1	2	3	4	5	6	7	8	9
Actual Heap										

Abstract Heap

Done!!

Quicksort

- Quicksort: the name says it all!
- It is the fastest algorithm that uses no extra space.
- It can also be optimised to be very, very fast indeed.
- It is $O(n\log n)$ on average and $O(n^2)$ in the worst case.
- *But* it is difficult to code and difficult to understand unless you actually try it.

Murdoch University

Quicksort Algorithm

```
•    QuickSort
•        Quicksort (0, size, array);
•    END QuickSort

•    QuickSort (low, high, array)
•        IF low < high AND high-low >= 2
•            integer pivotIndex
•            Split (low, high, array, pivotIndex) // sort is
     done here
•            QuickSort (low, pivotIndex-1, array)
•            QuickSort (pivotIndex+1, high, array)
•        ELSEIF high-low == 2
•            If array[high] < array[low]
•                Swap them
•            ENDIF
•        ENDIF
•    END QuickSort
```

Murdoch UNIVERSITY

```
Split (low, high, array, pivotIndex)
    pvalue = array[low]
    integer index1 = low
    integer index2 = high
    WHILE (index1 < index2)
        WHILE (array[index1] <= pvalue && index1 < index2)
          index1++;
        ENDWHILE
        WHILE (array[index2] > pvalue && index2 > index1)
          index2--;
        ENDWHILE
        IF (index1 < index2)
          Swap values at index1 and index2
        ENDIF
    ENDWHILE
    Set pivotIndex to index2-1
    Swap values at low and pivotIndex
End Split
```

Look for a value higher than the pivot value

Look for a value lower than the pivot value

If found, swap them

Now put the pivot value between them

Murdoch UNIVERSITY

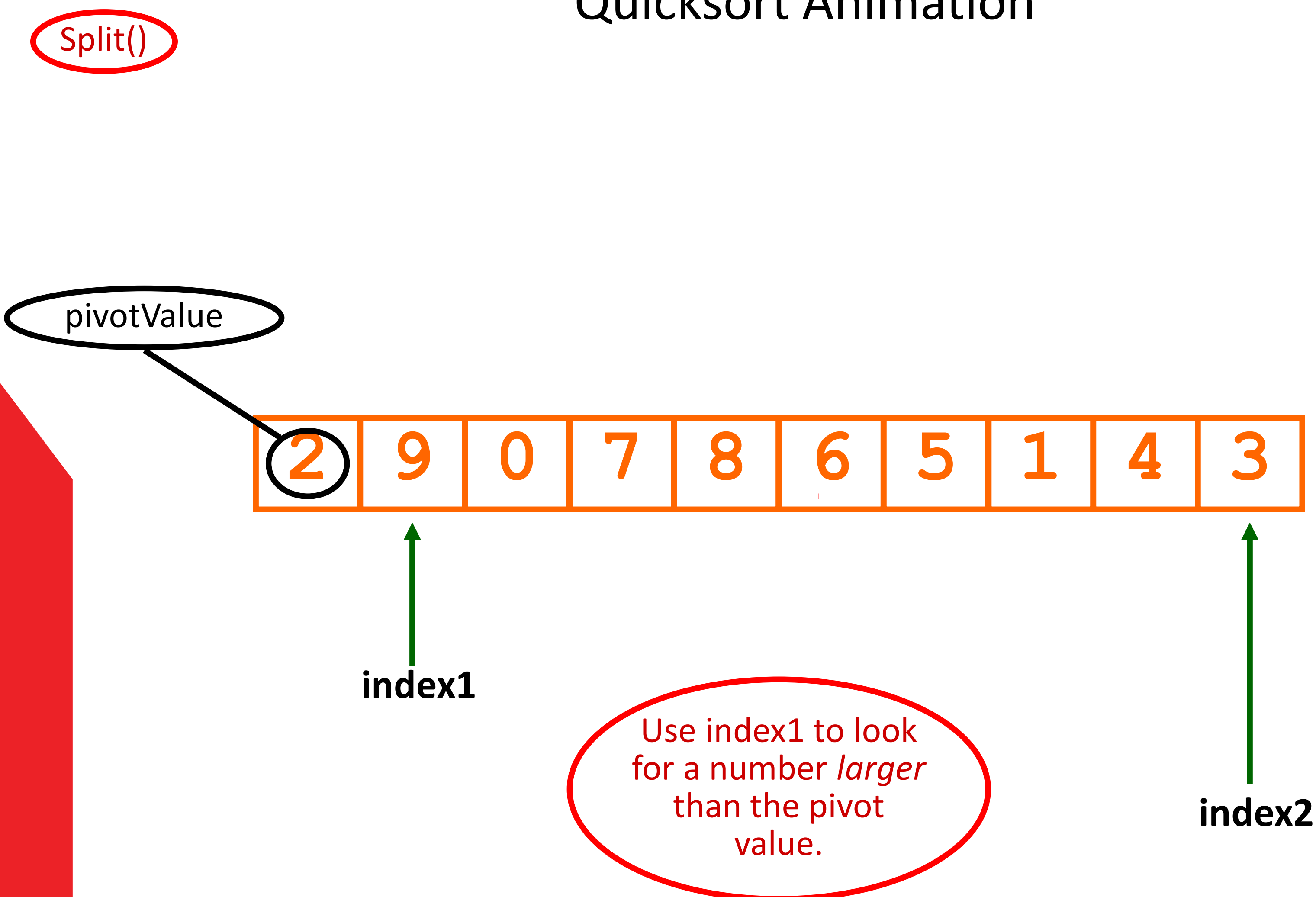
Quicksort Animation

Split()

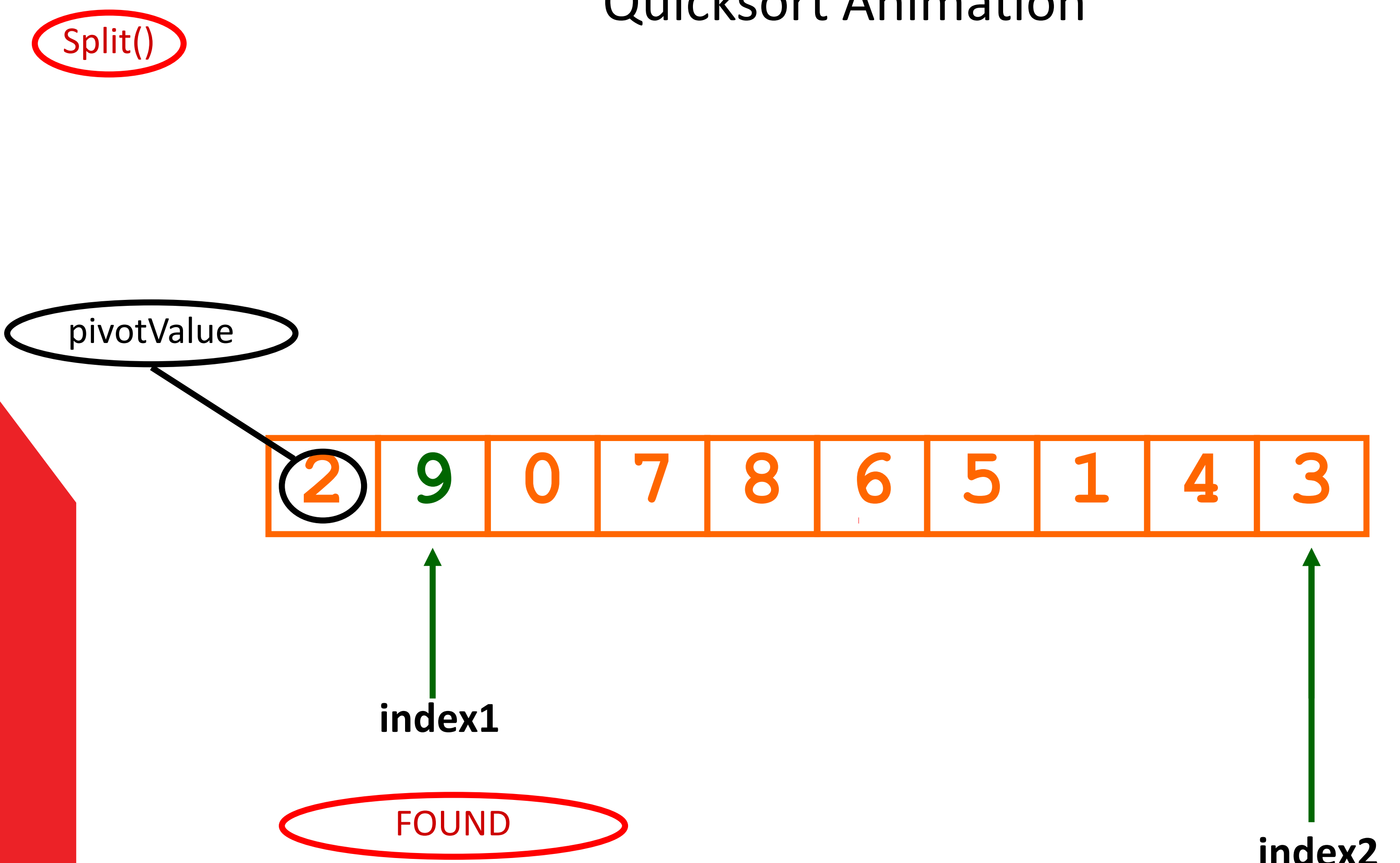
pivotValue

2	9	0	7	8	6	5	1	4	3

Quicksort Animation

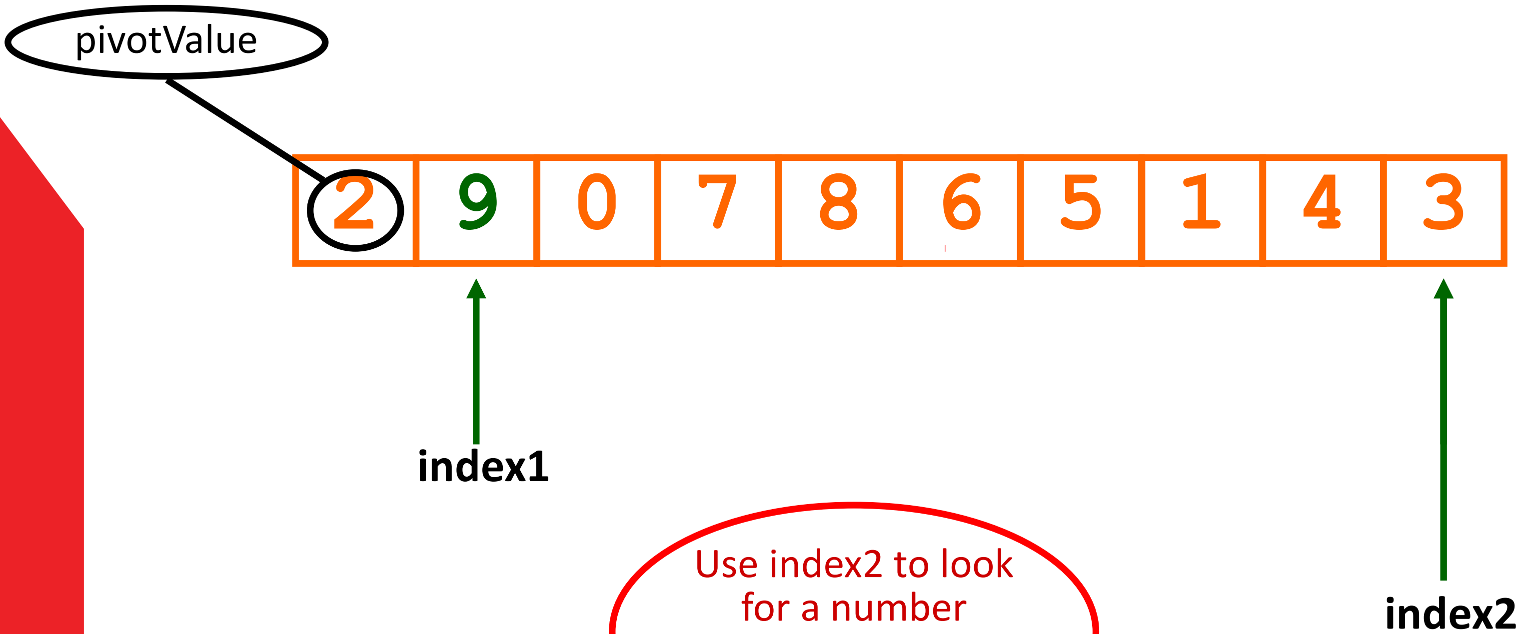


Quicksort Animation



Quicksort Animation

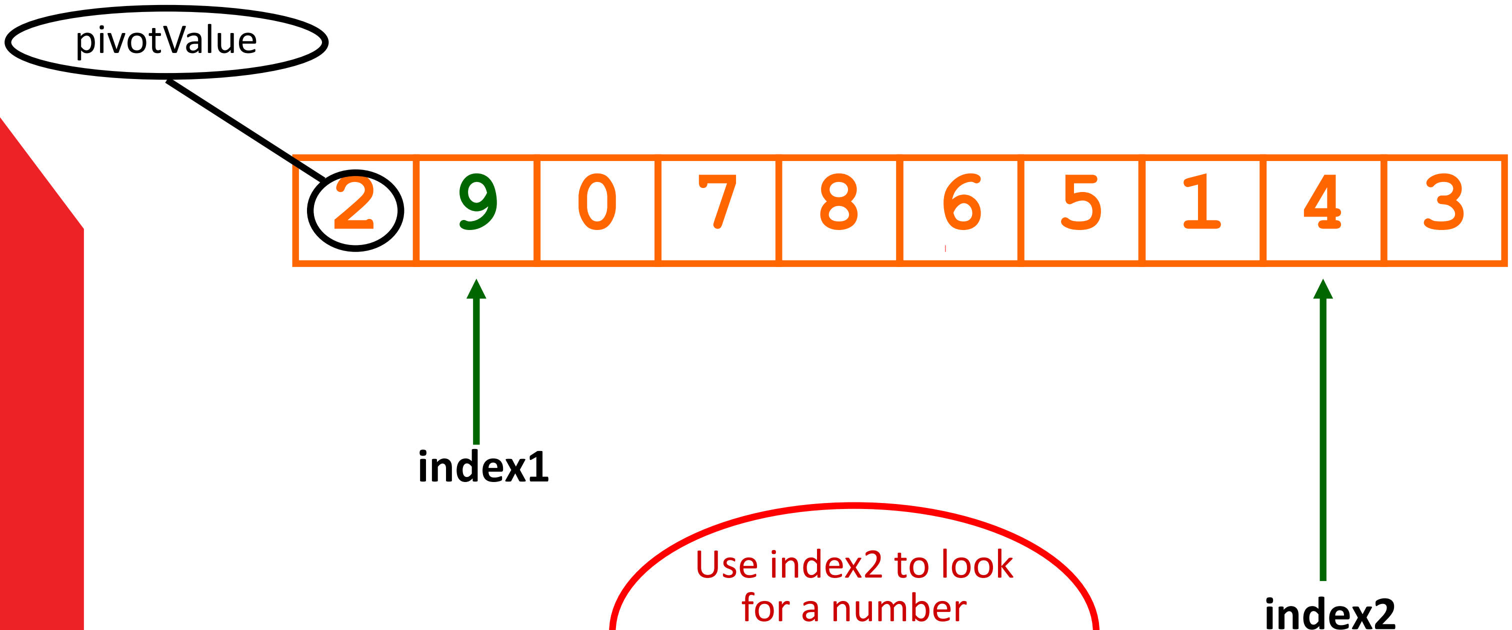
Split()



Use index2 to look for a number *smaller* than the pivot value.

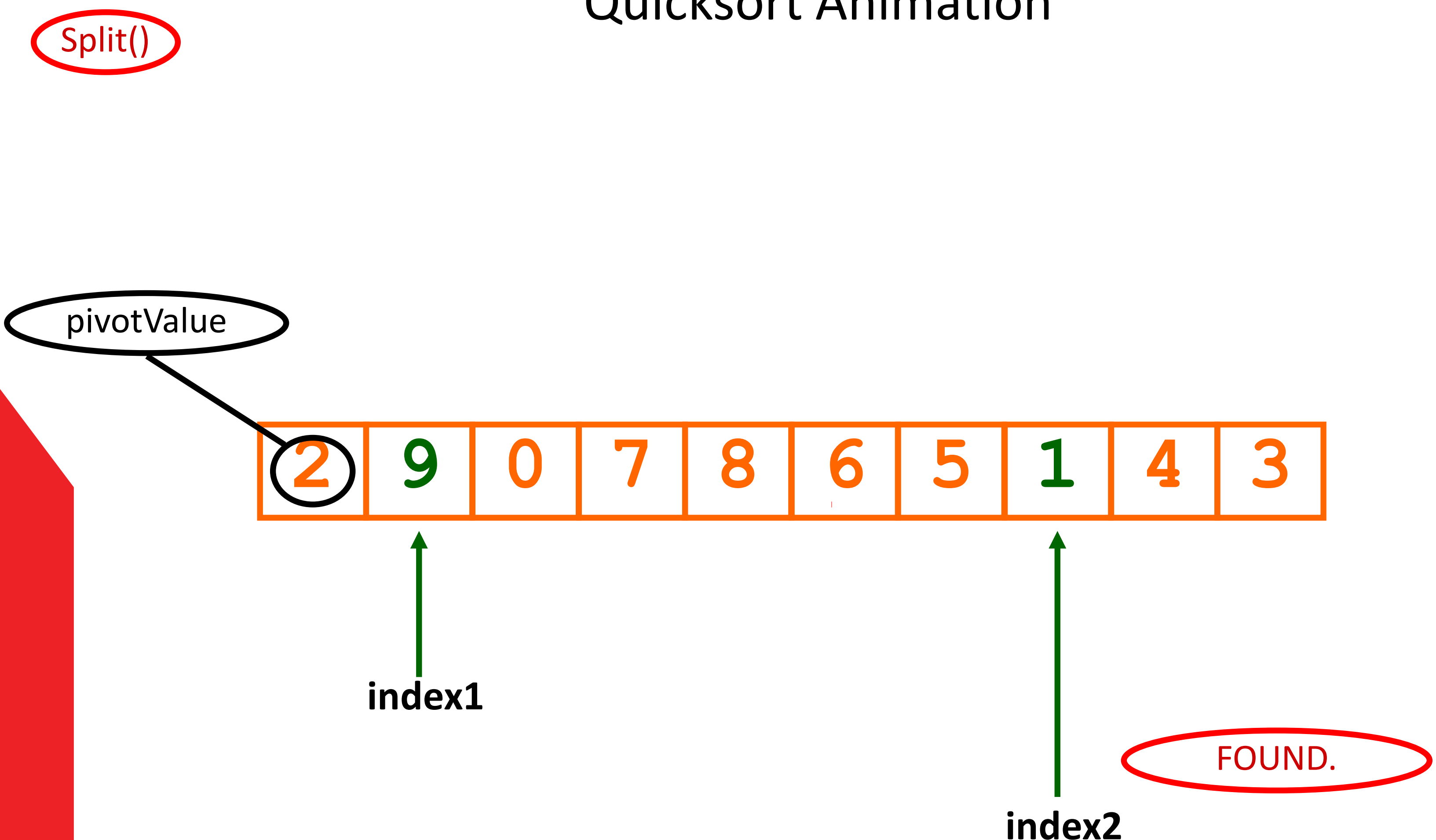
Quicksort Animation

Split()



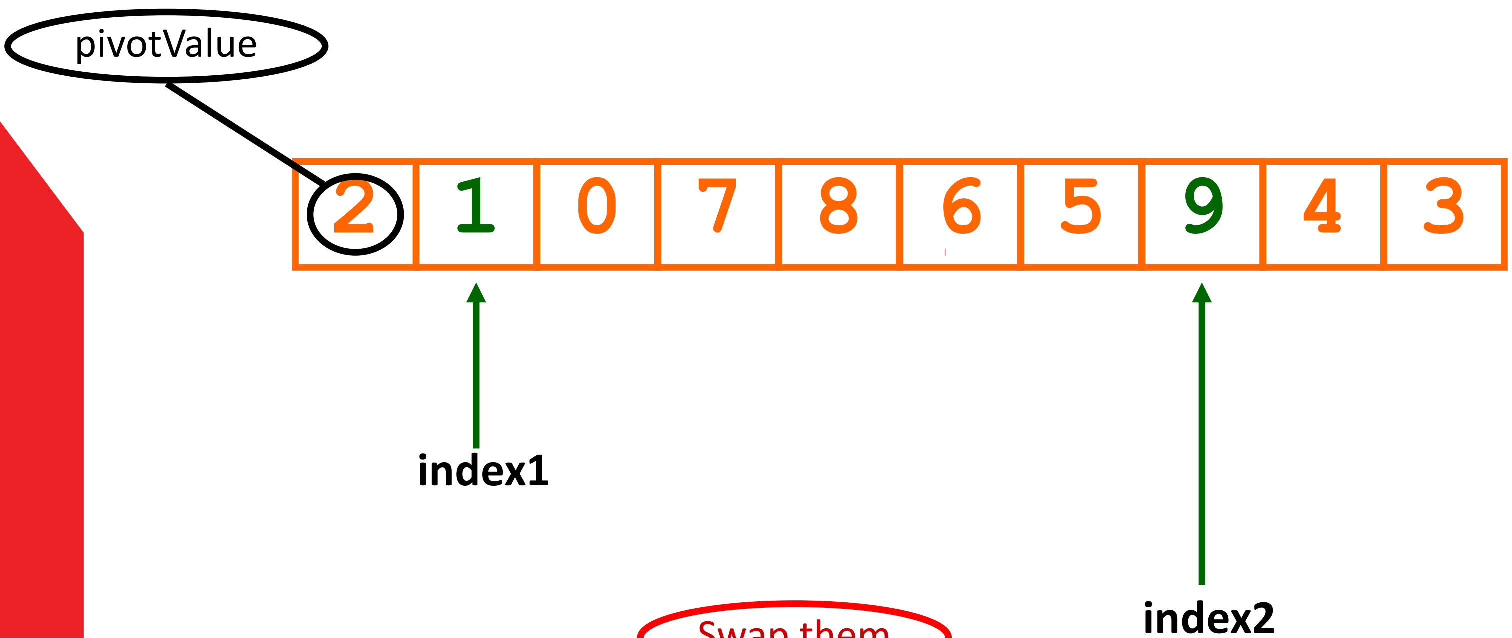
Use index2 to look for a number *smaller* than the pivot value.

Quicksort Animation



Quicksort Animation

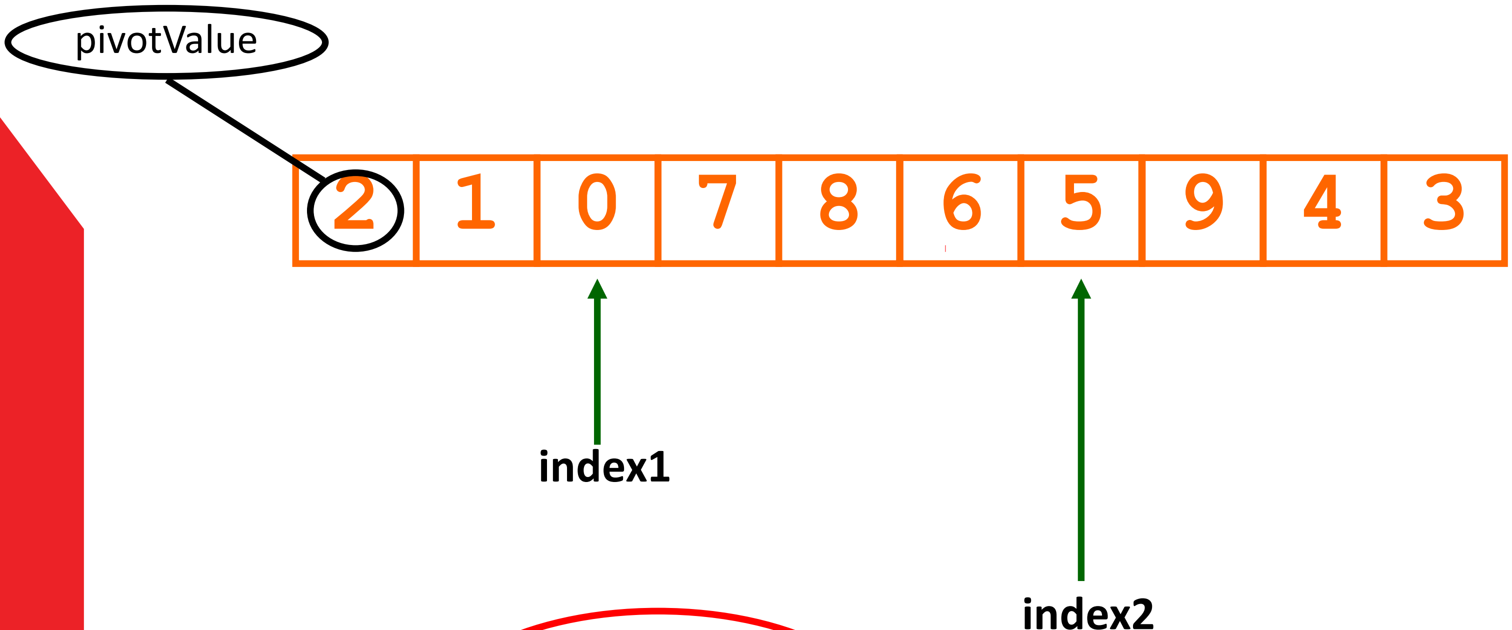
Split()



Swap them

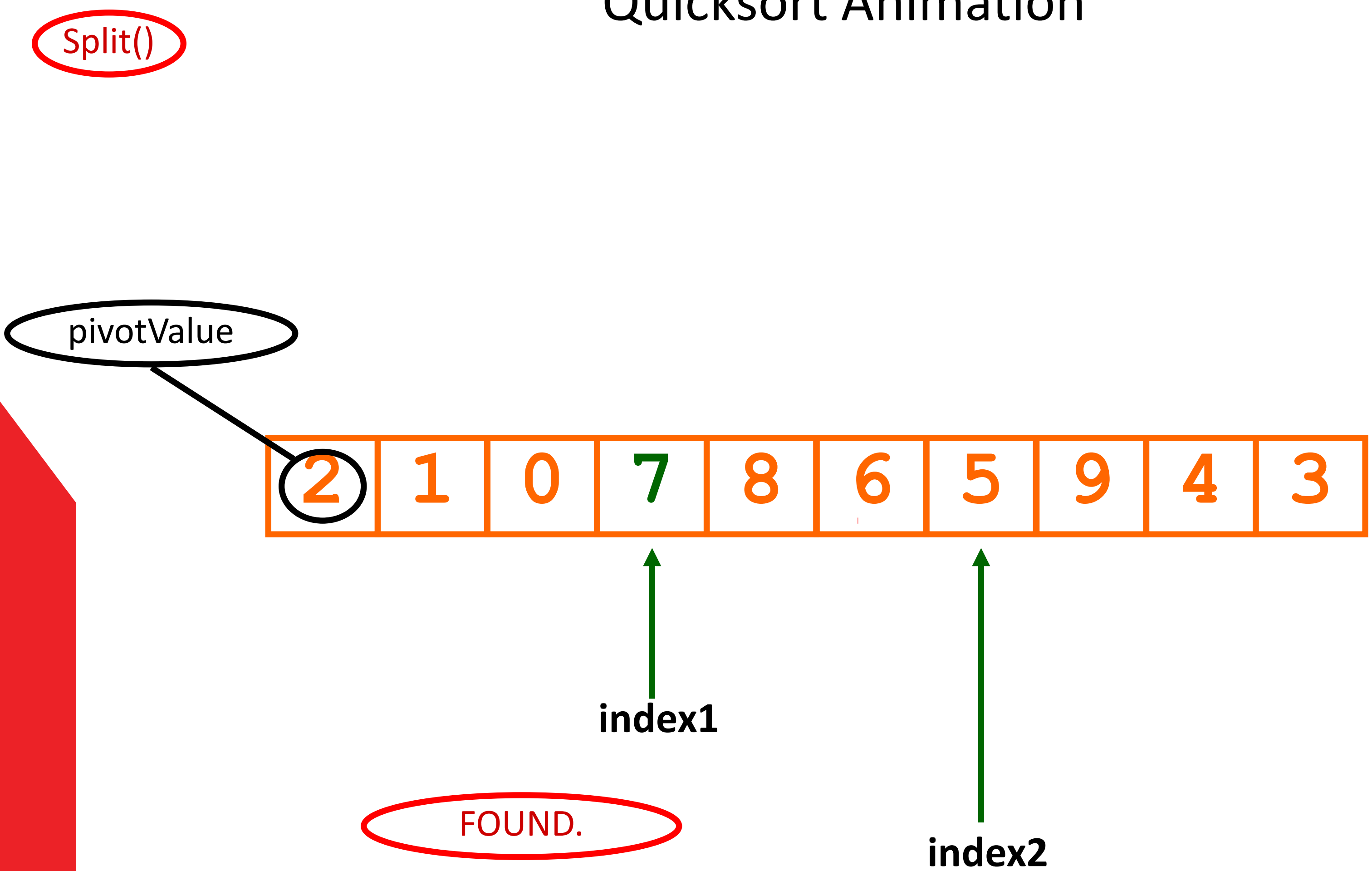
Quicksort Animation

Split()



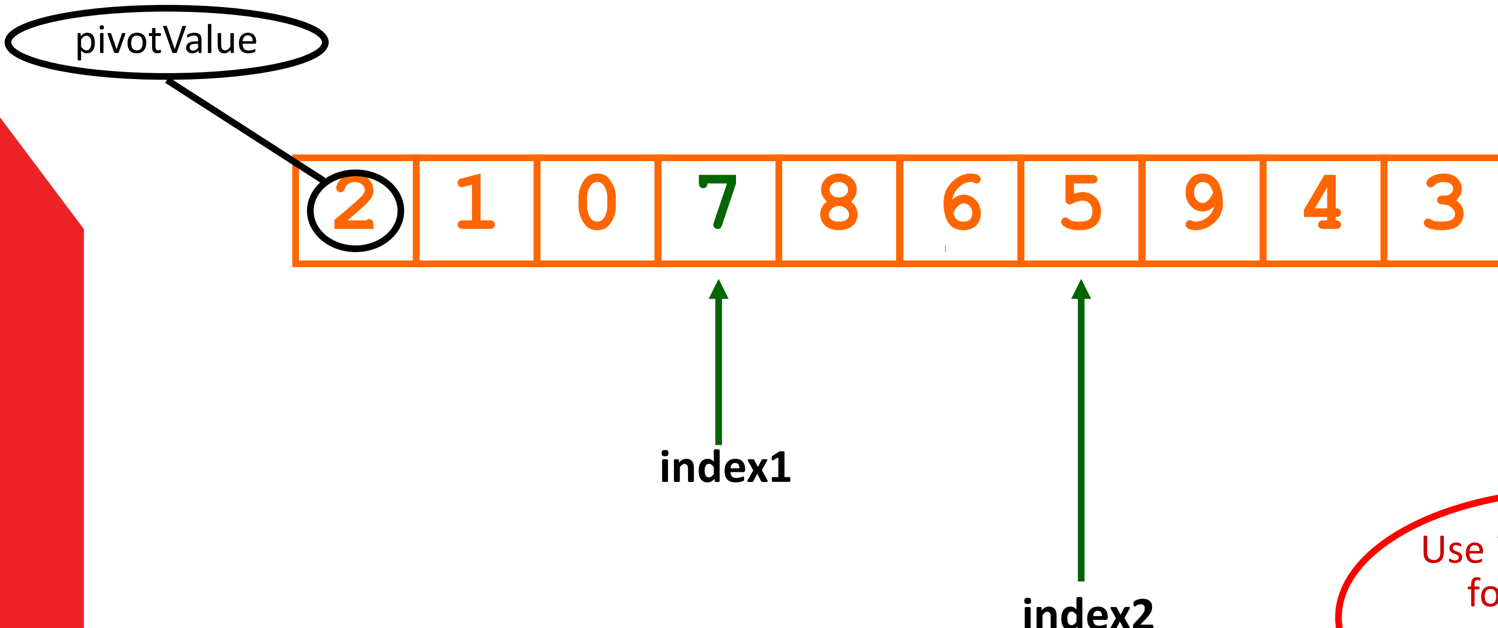
Use index1 to look for a number *larger* than the pivot value.

Quicksort Animation



Quicksort Animation

Split()



Use index2 to look for a number *smaller* than the pivot value.

Quicksort Animation

Split()

pivotValue

2	1	0	7	8	6	5	9	4	3

index1

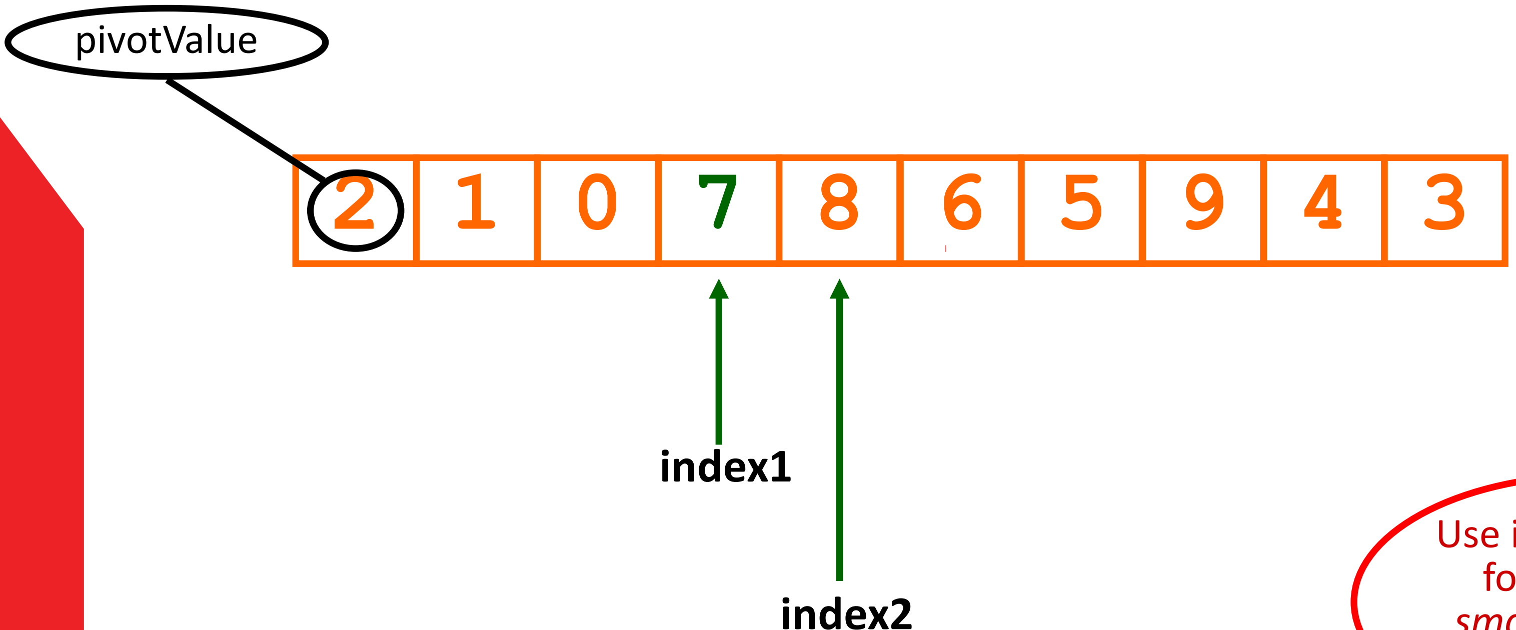
index2

Use index2 to look for a number *smaller* than the pivot value.

Murdoch UNIVERSITY

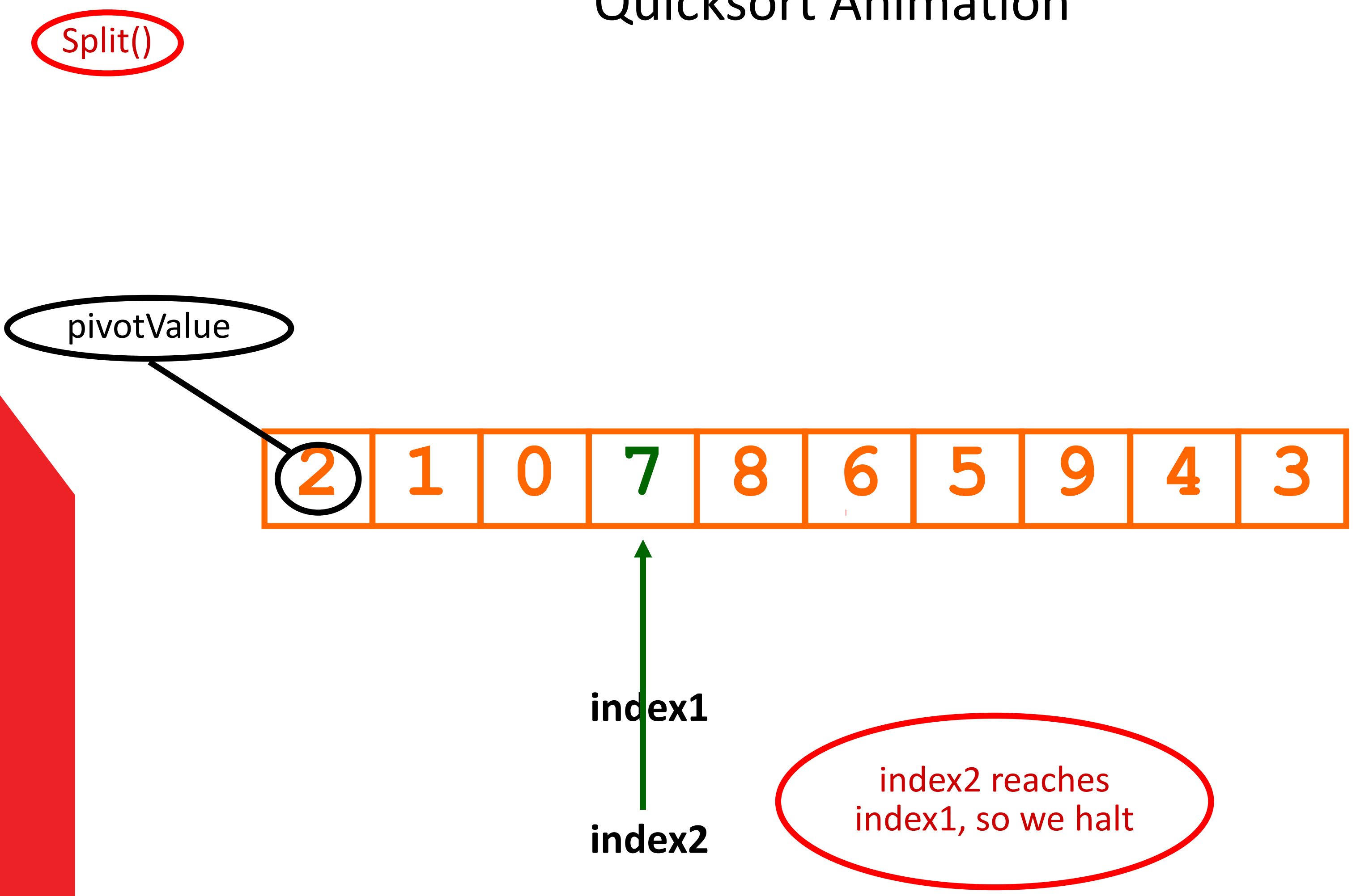
Quicksort Animation

Split()

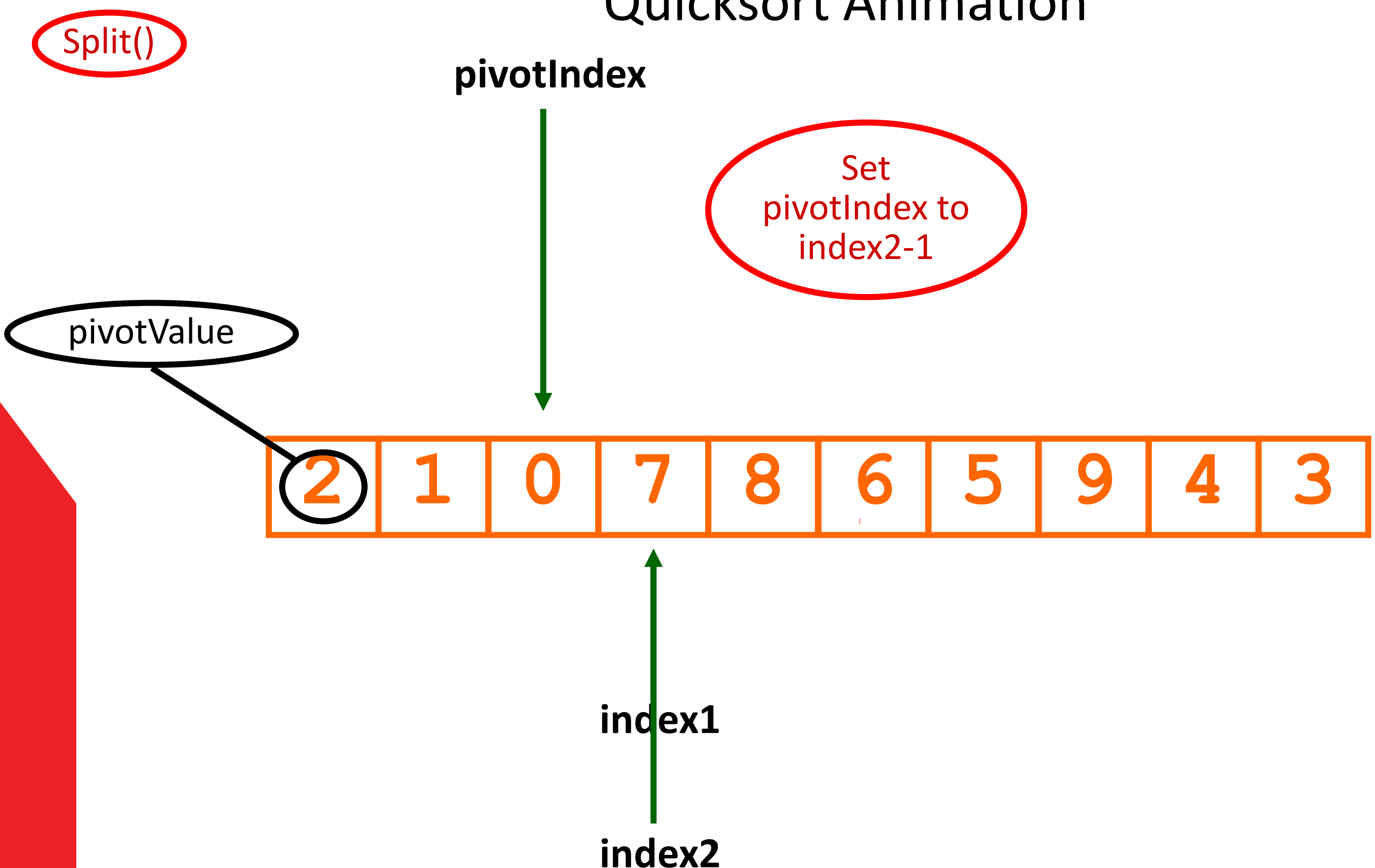


Use index2 to look for a number *smaller* than the pivot value.

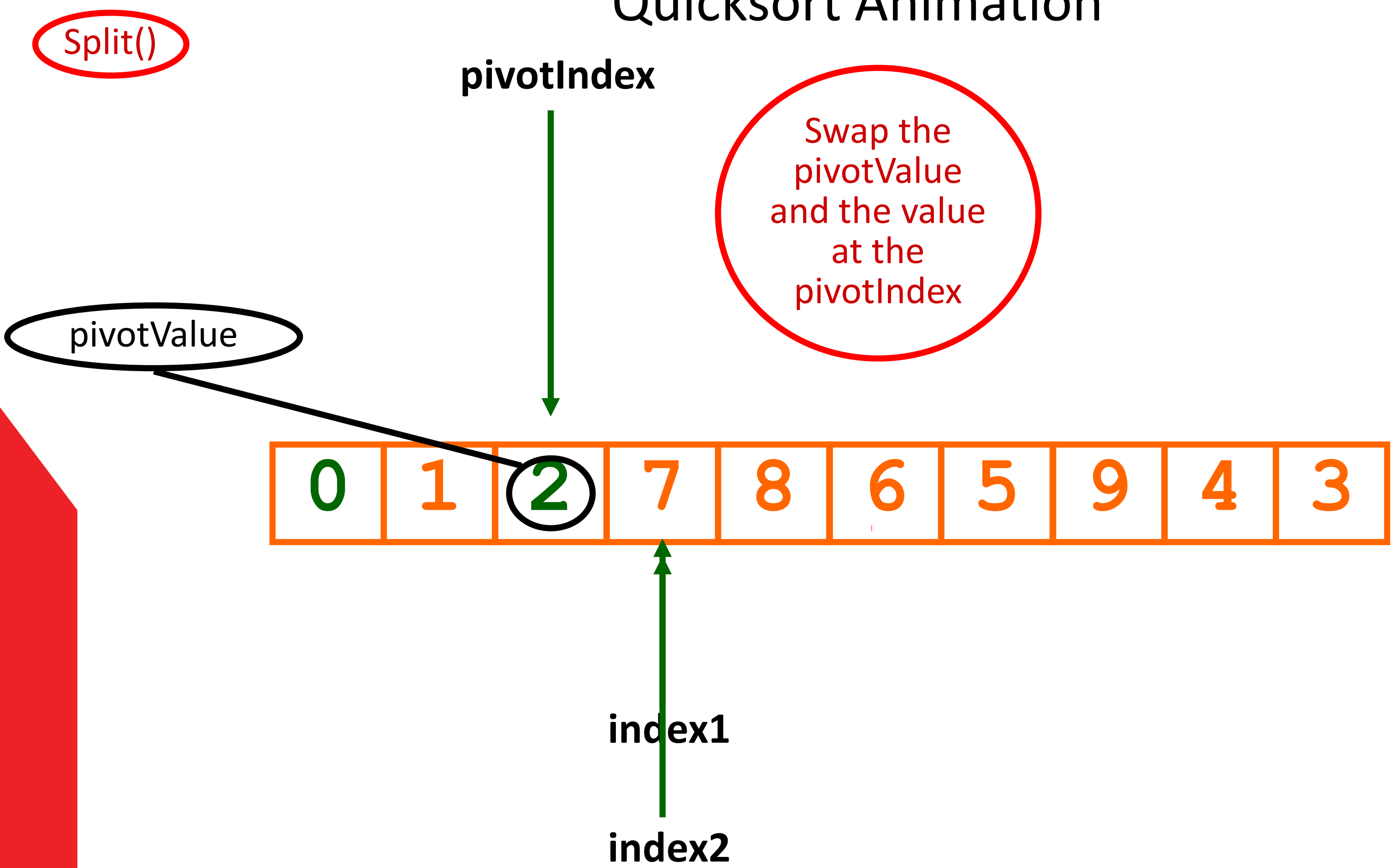
Quicksort Animation



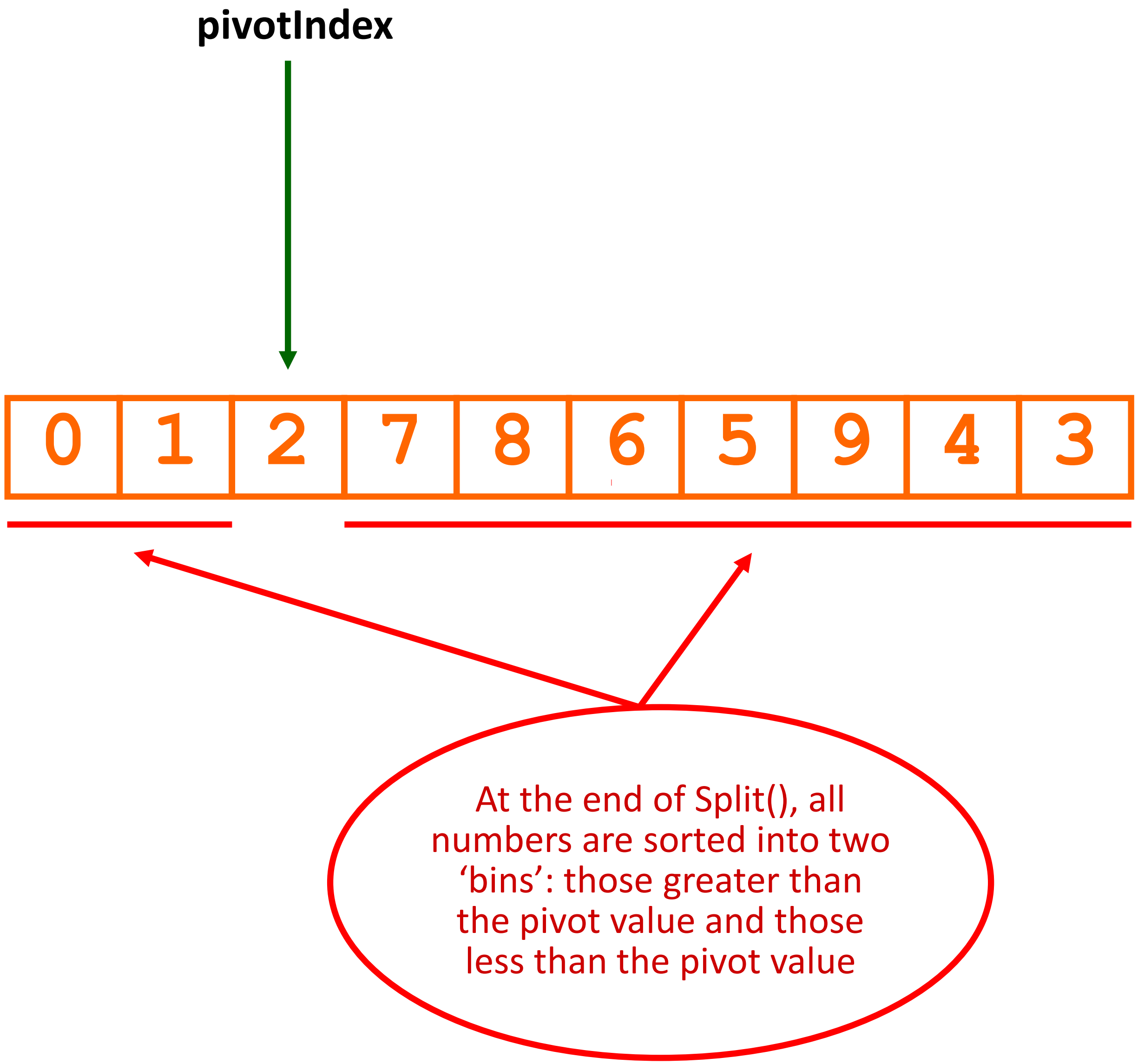
Quicksort Animation



Quicksort Animation



Quicksort Animation



Murdoch UNIVERSITY

Quicksort Animation

pivotIndex

0	1	2	7	8	6	5	9	4	3

Quicksort the section below pivotIndex

Murdoch UNIVERSITY

Quicksort Animation

pivotIndex

0	1	2	7	8	6	5	9	4	3

Less than three elements and already in order

Murdoch UNIVERSITY

Quicksort Animation

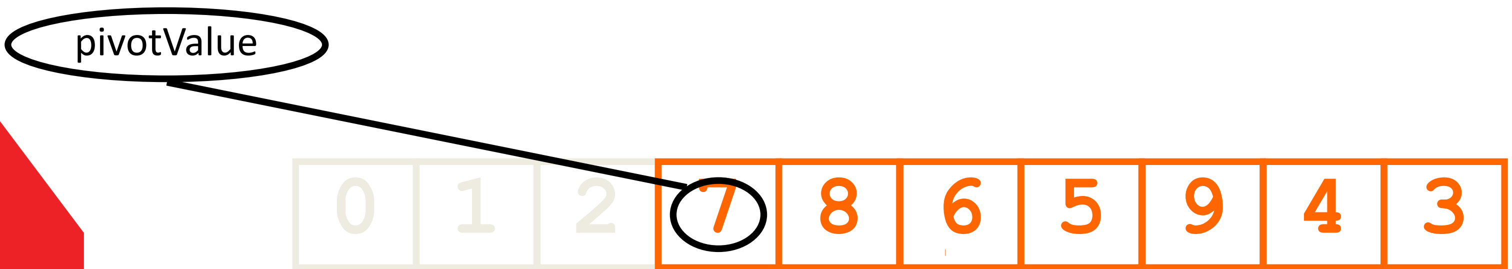
pivotIndex

0	1	2	7	8	6	5	9	4	3

Quicksort the section above pivotIndex

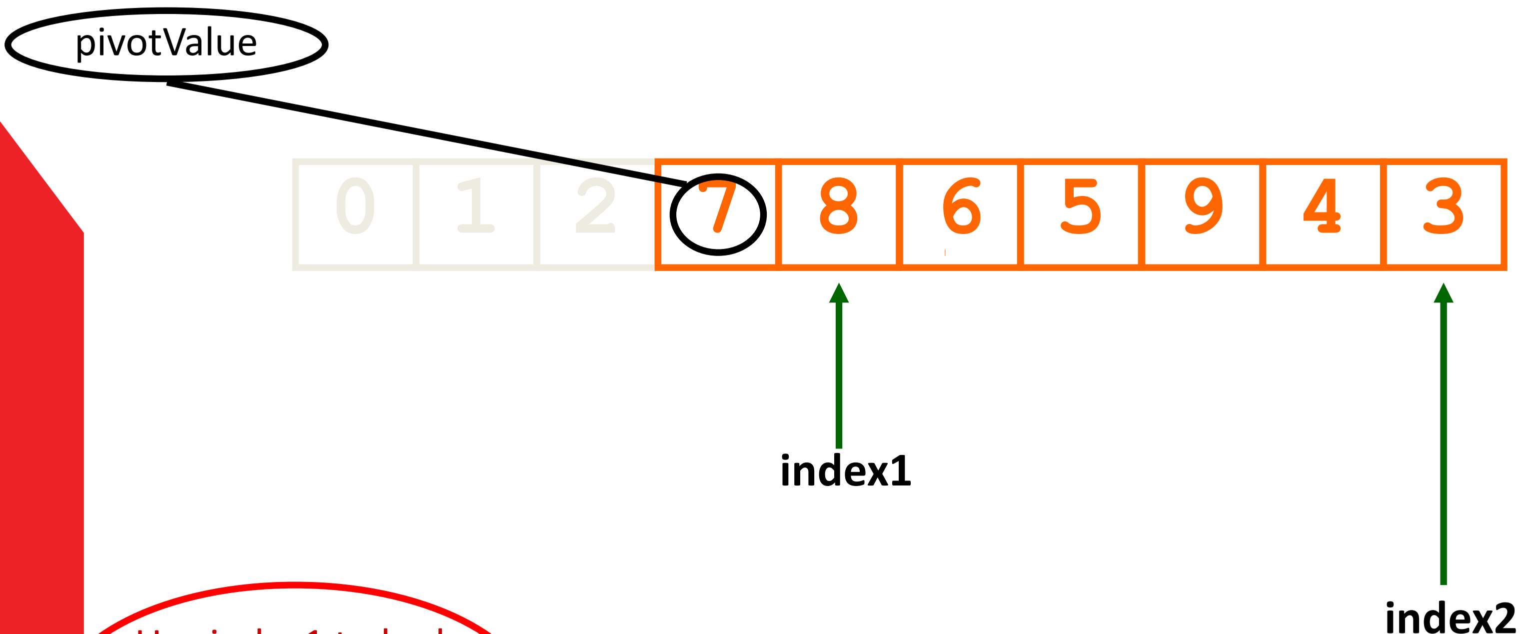
Quicksort Animation

Split()



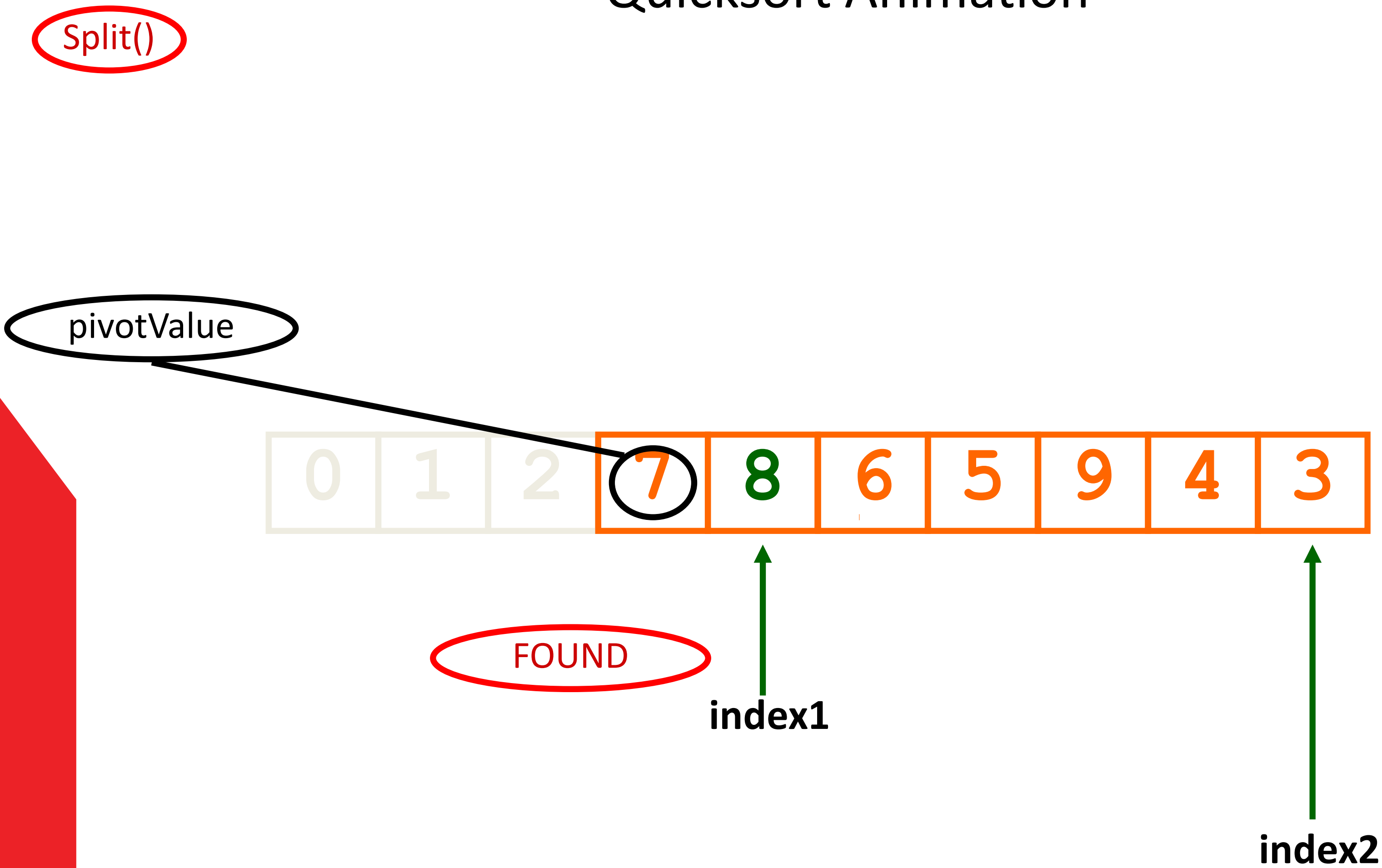
Quicksort Animation

Split()

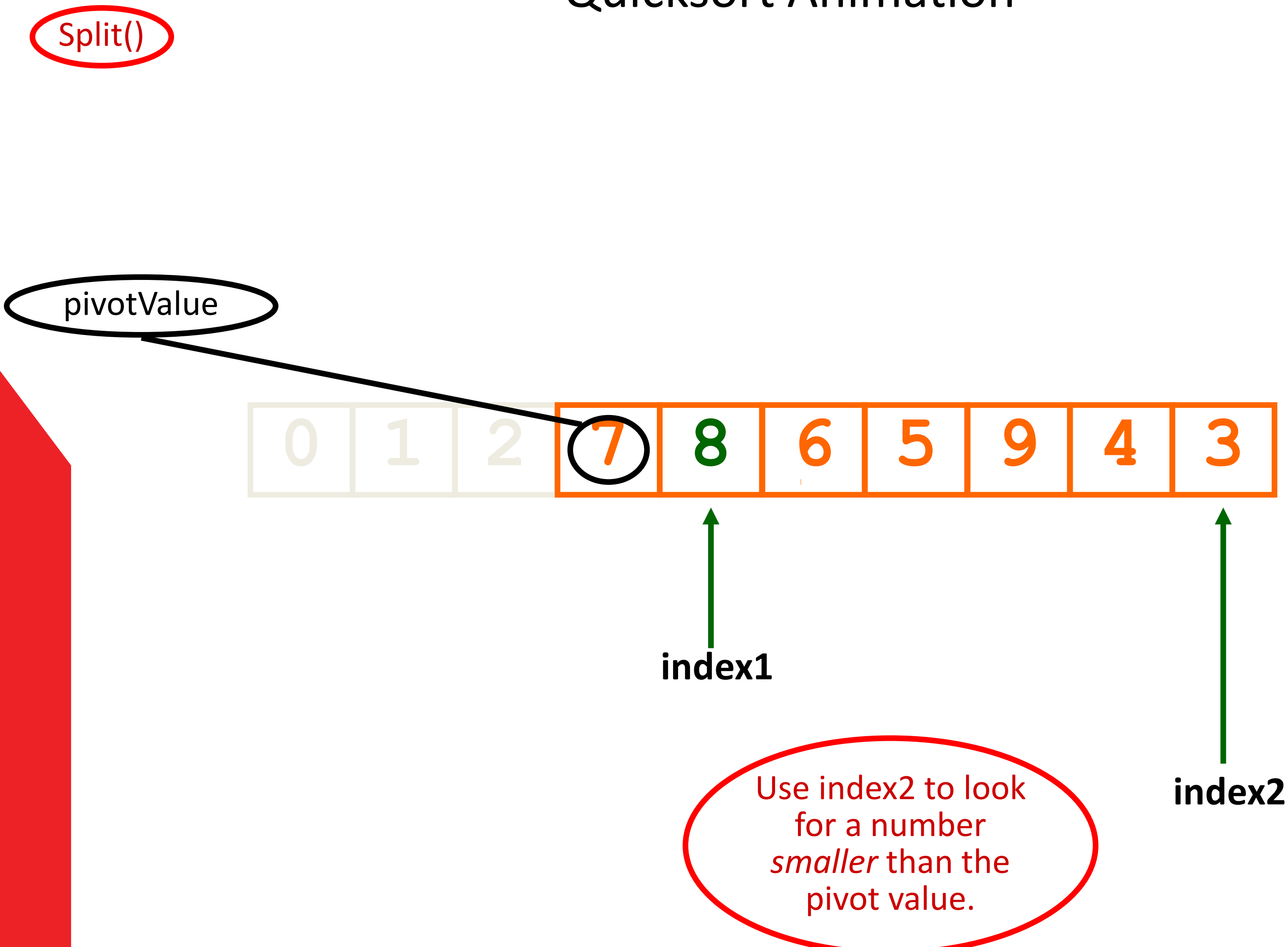


Use index1 to look for a number *larger* than the pivot value.

Quicksort Animation

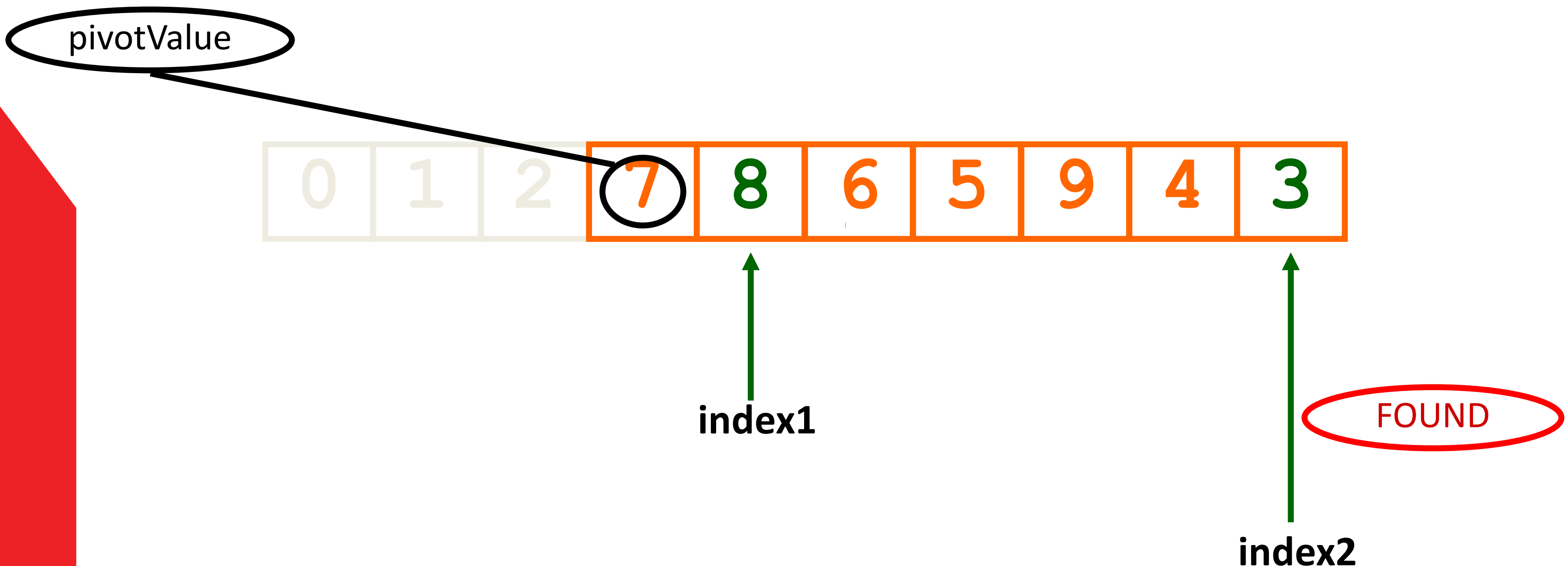


Quicksort Animation

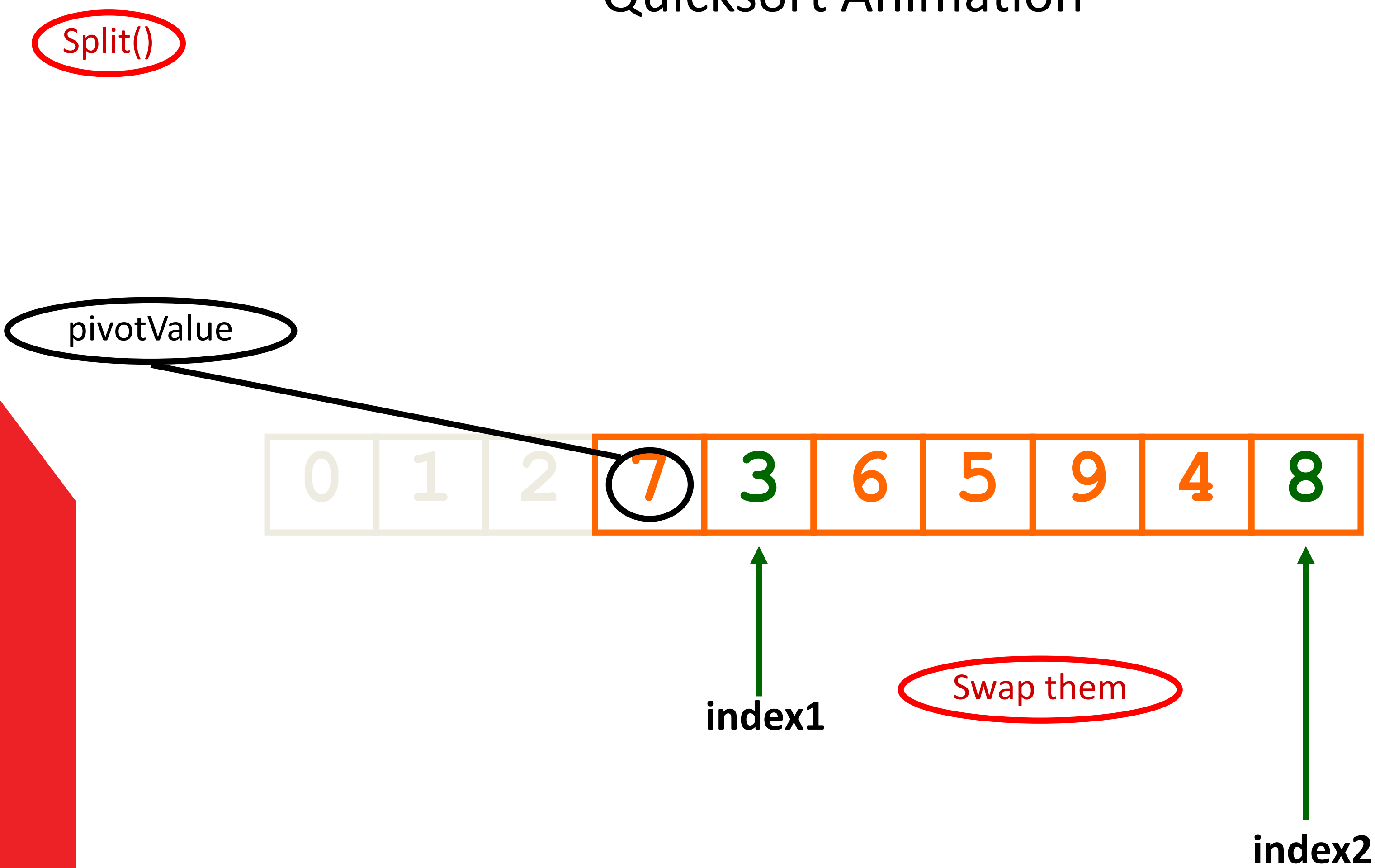


Quicksort Animation

Split()

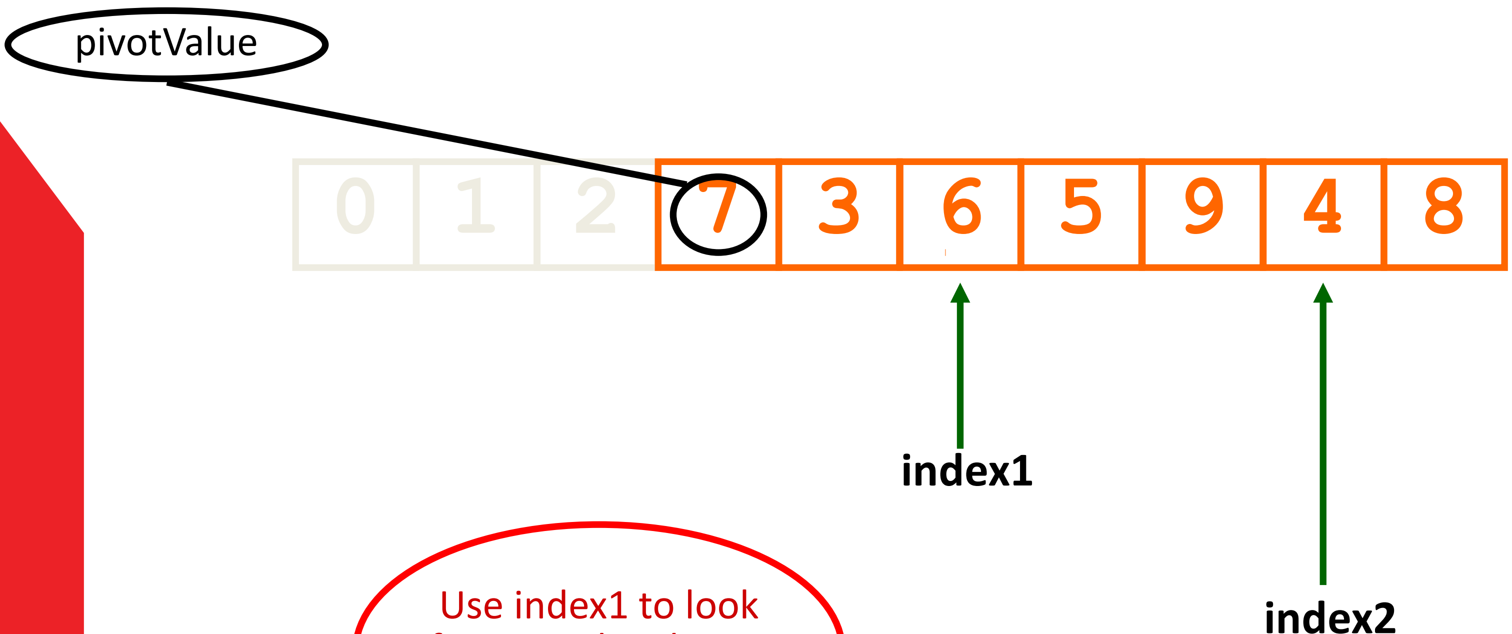


Quicksort Animation



Quicksort Animation

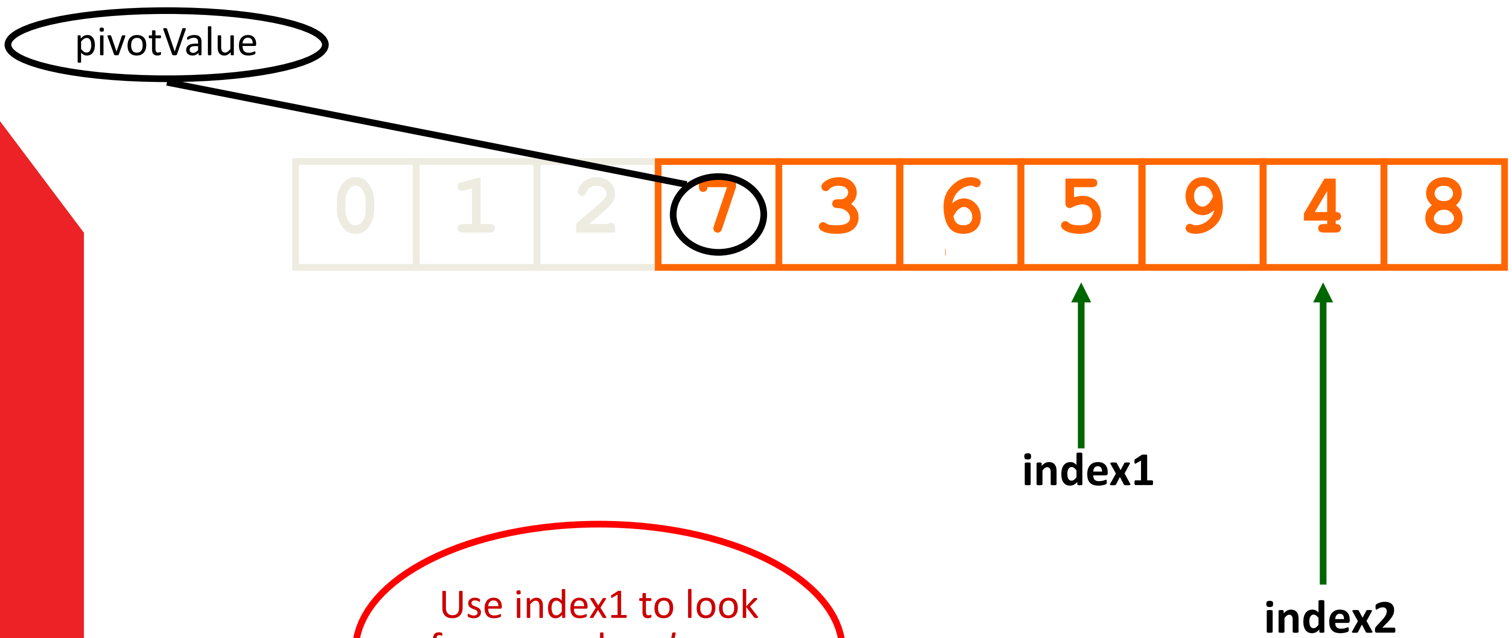
Split()



Use index1 to look for a number *larger* than the pivotValue

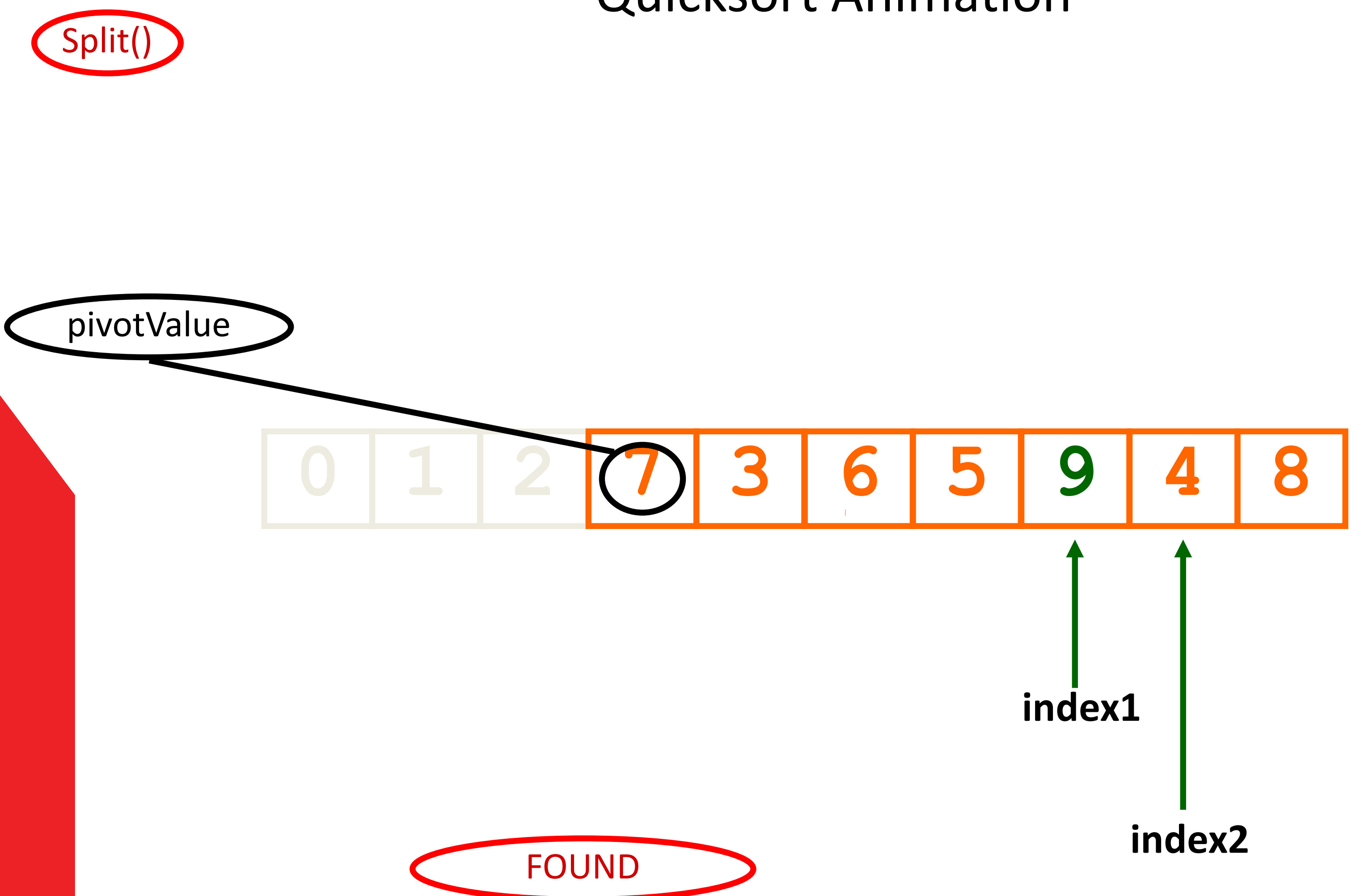
Quicksort Animation

Split()



Use index1 to look for a number *larger* than the pivotValue

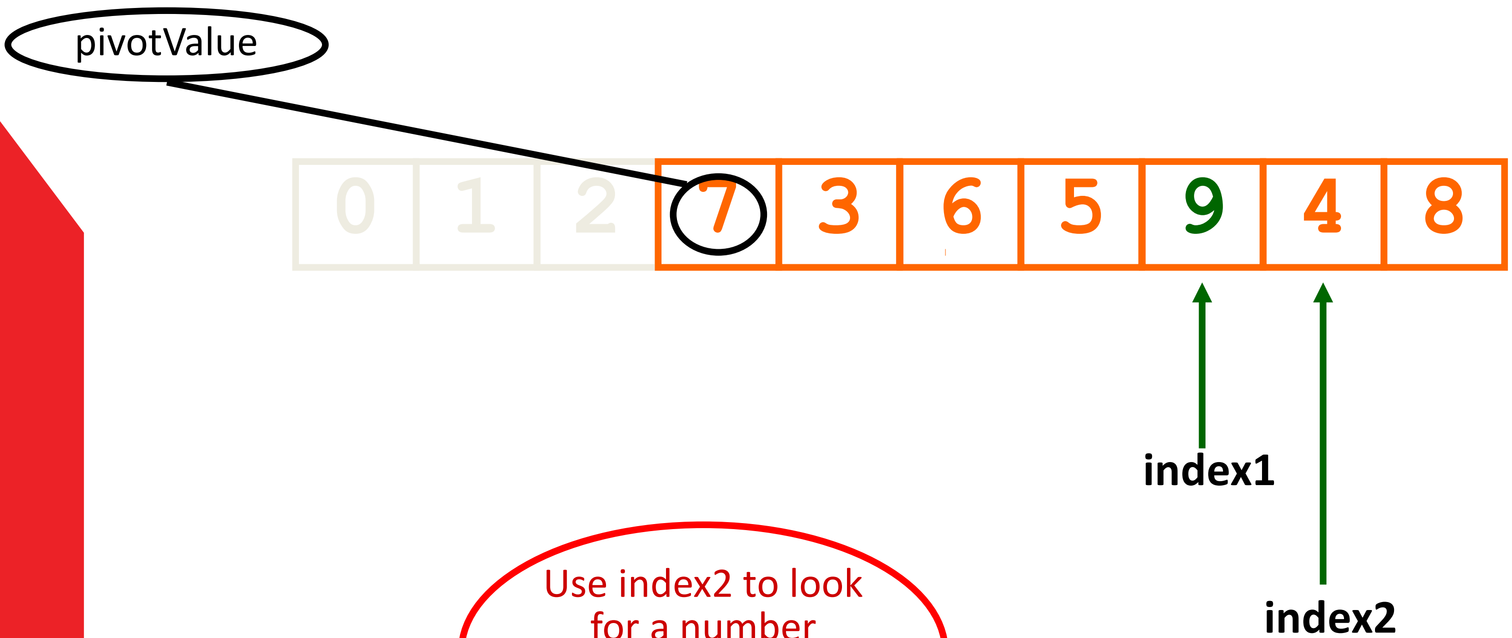
Quicksort Animation



Murdoch UNIVERSITY

Quicksort Animation

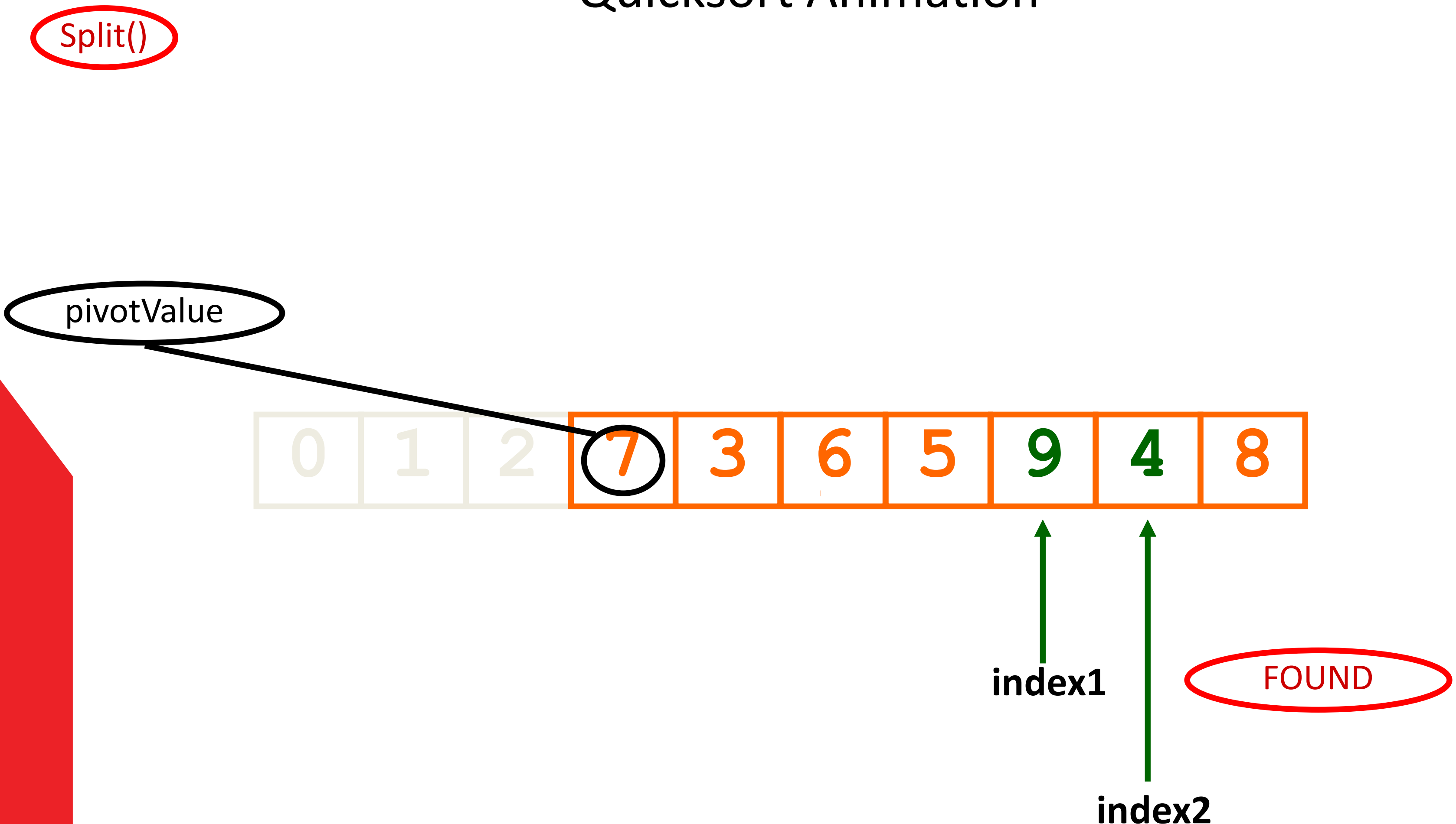
Split()



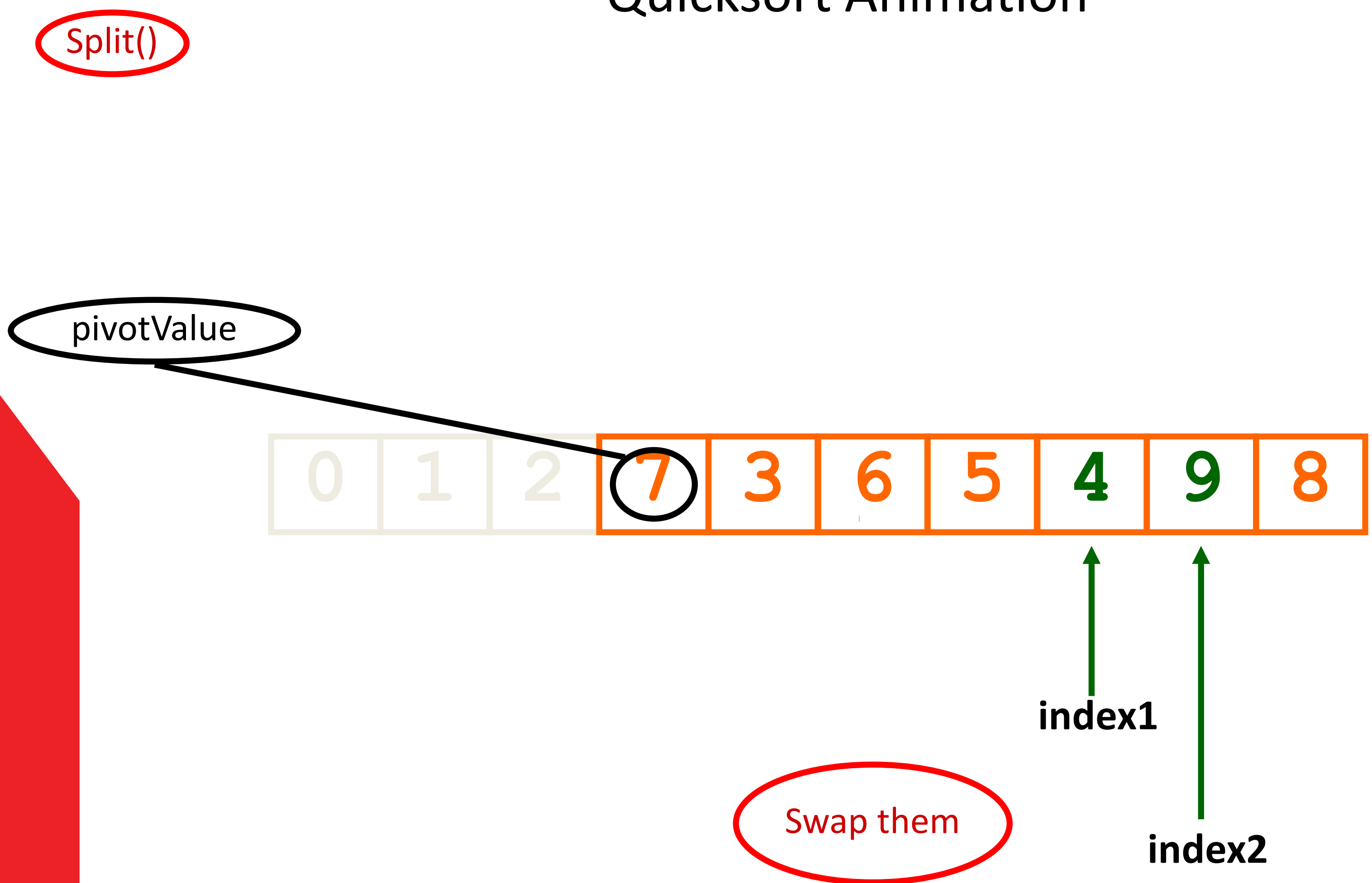
Use index2 to look for a number *smaller* than the pivot value

Murdoch University

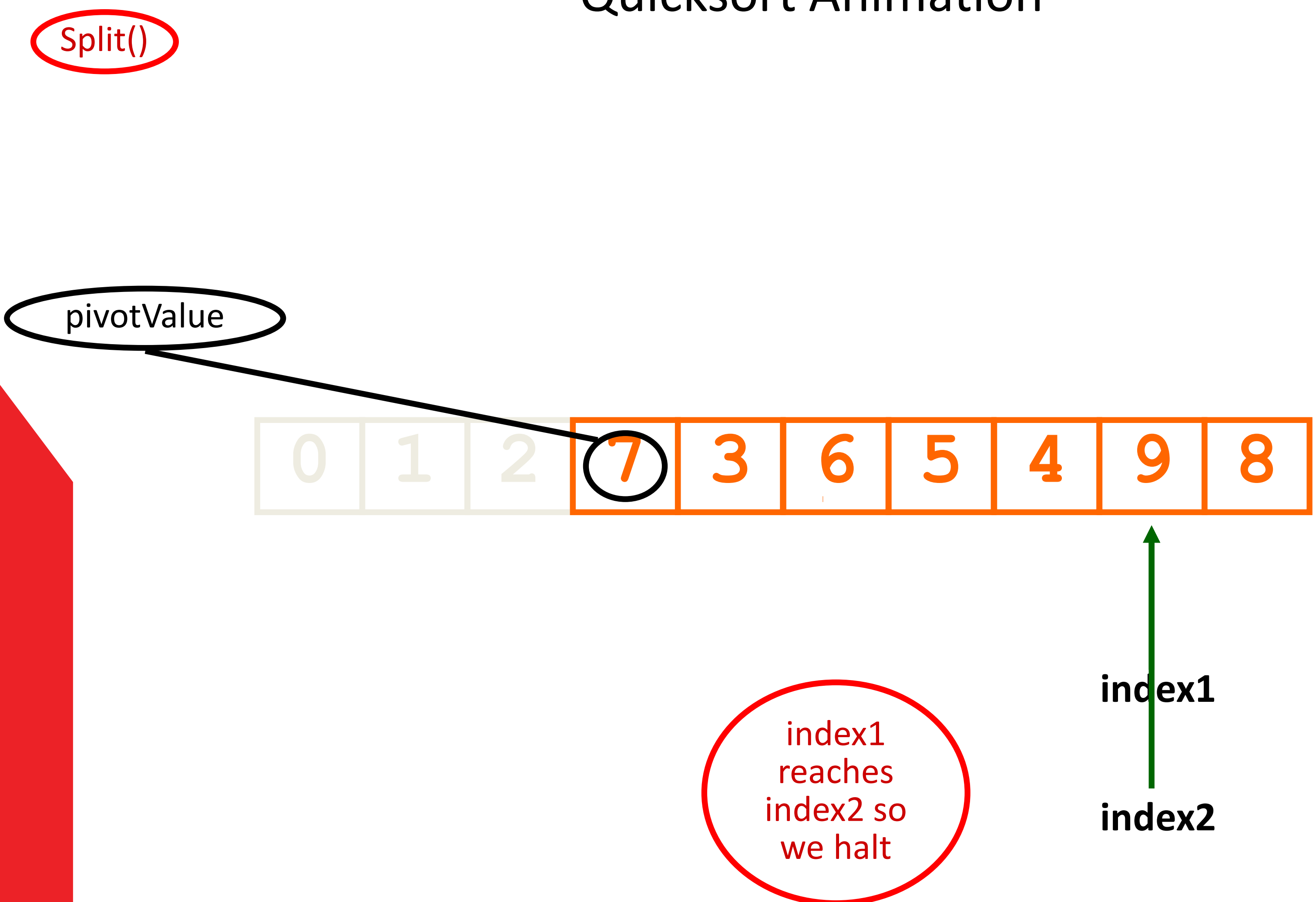
Quicksort Animation



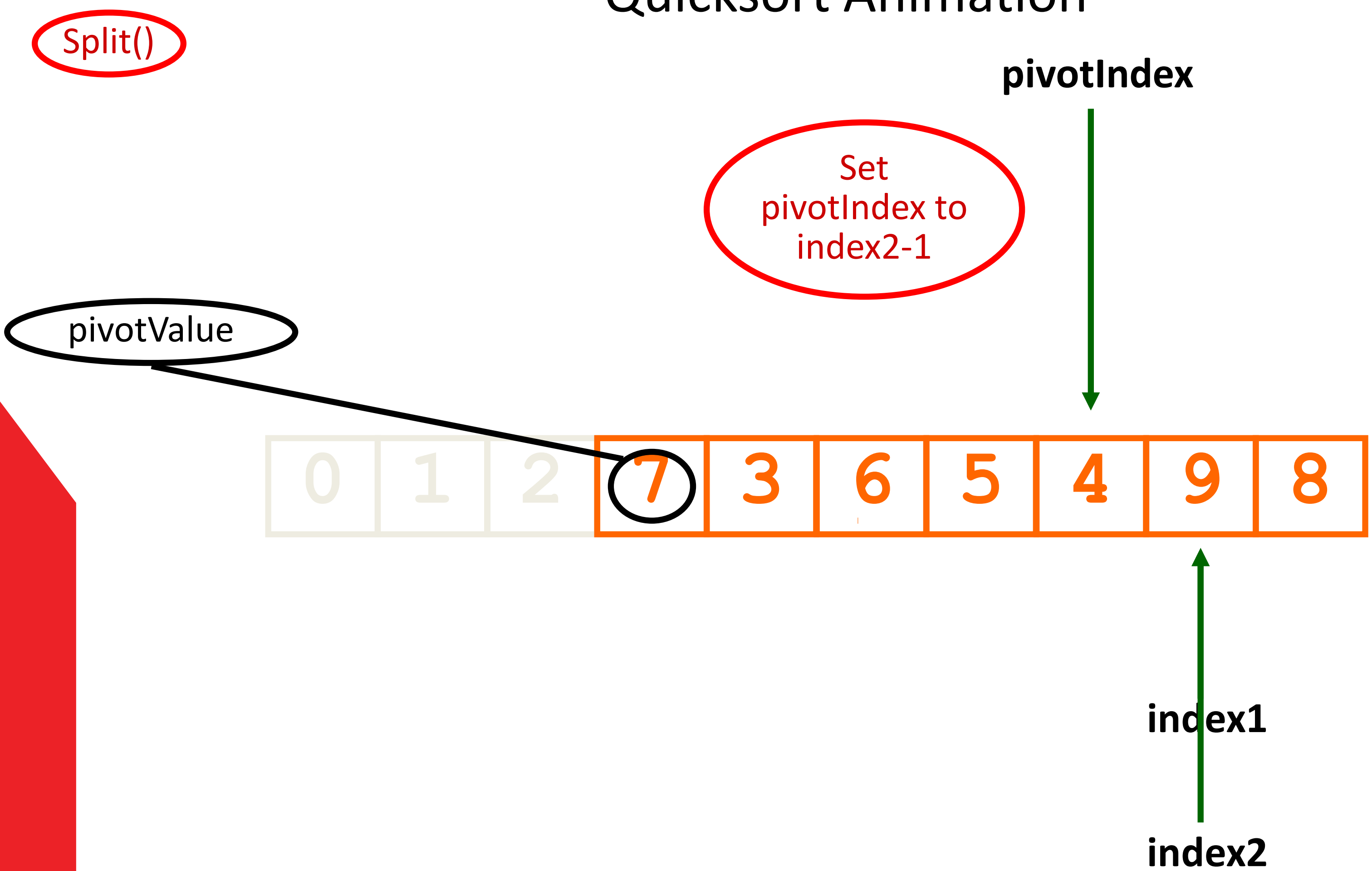
Quicksort Animation



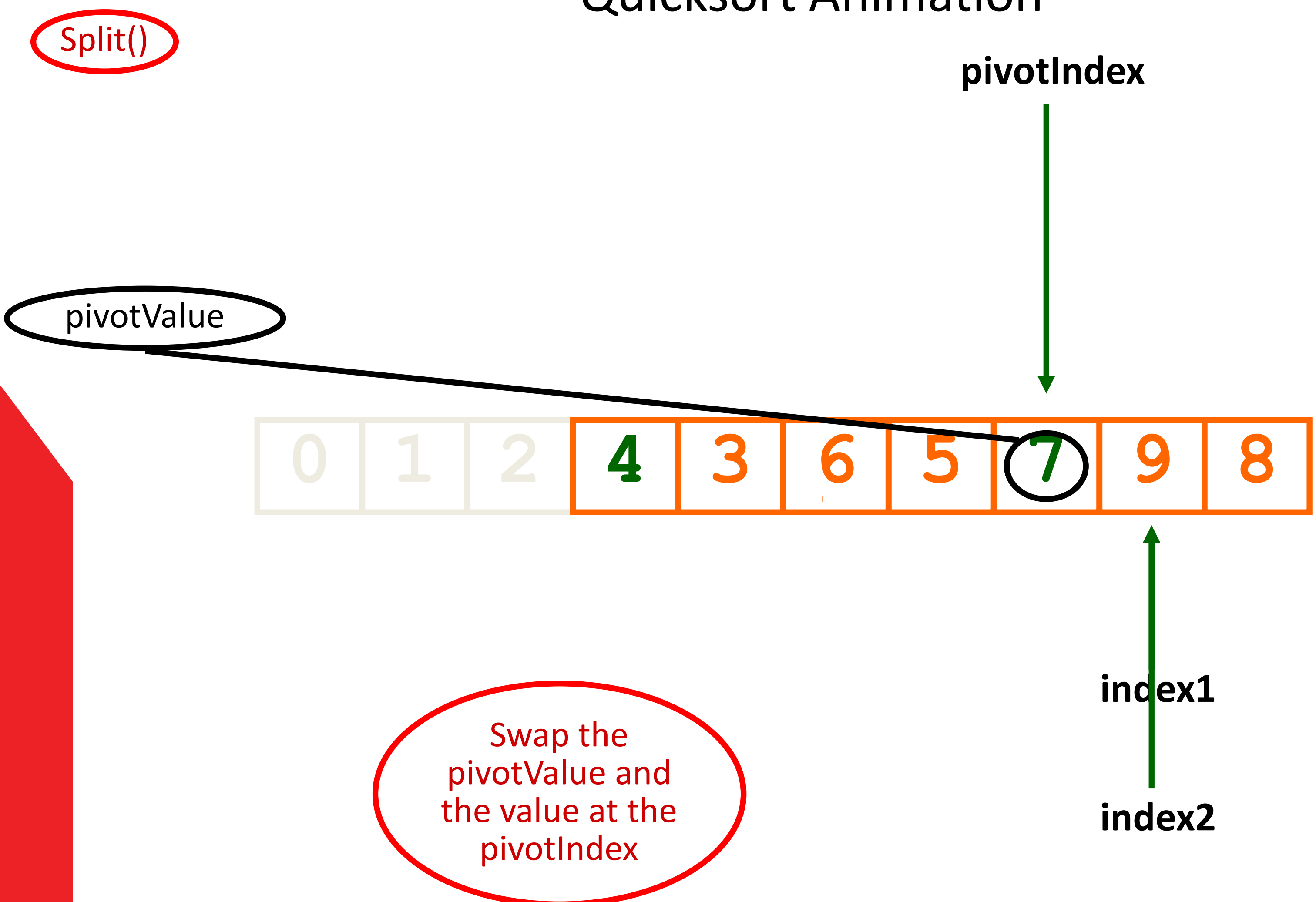
Quicksort Animation



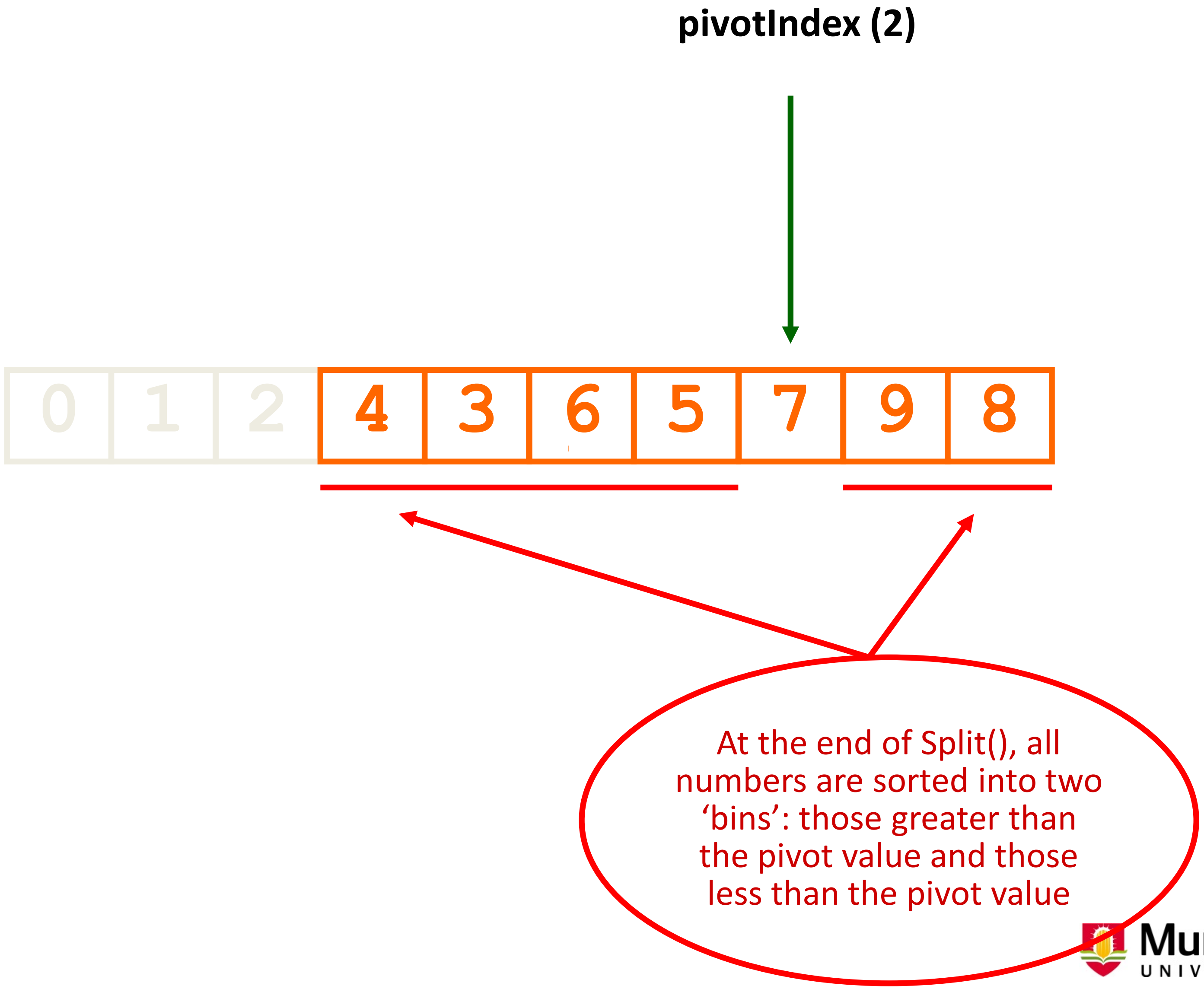
Quicksort Animation



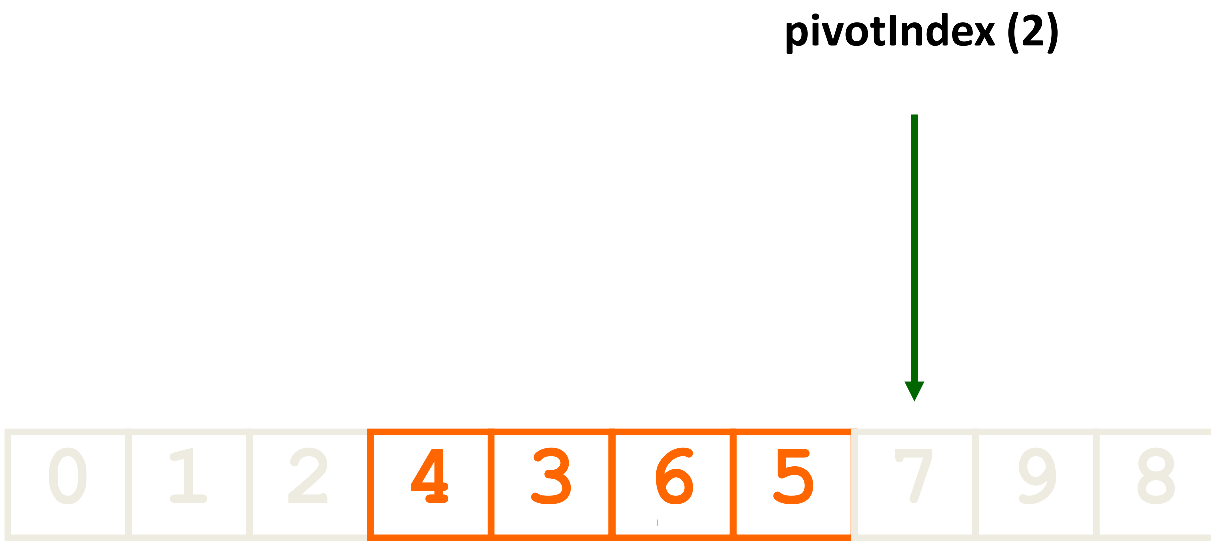
Quicksort Animation



Quicksort Animation

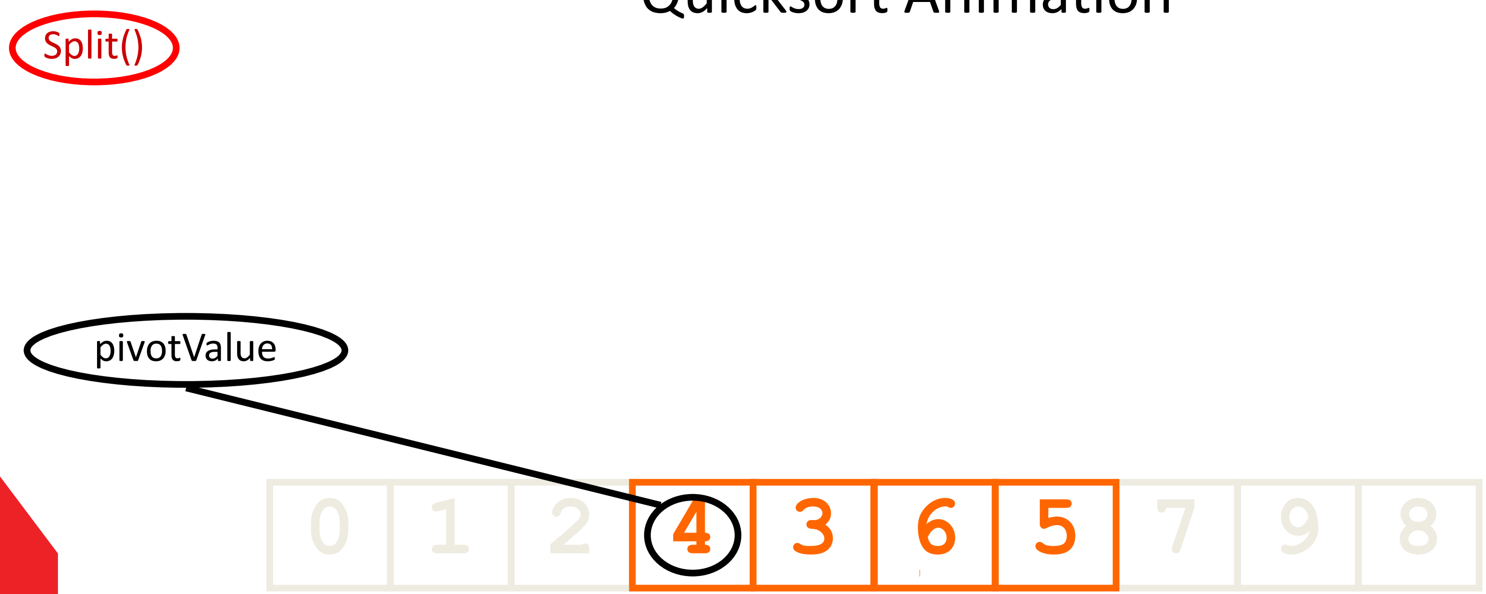


Quicksort Animation

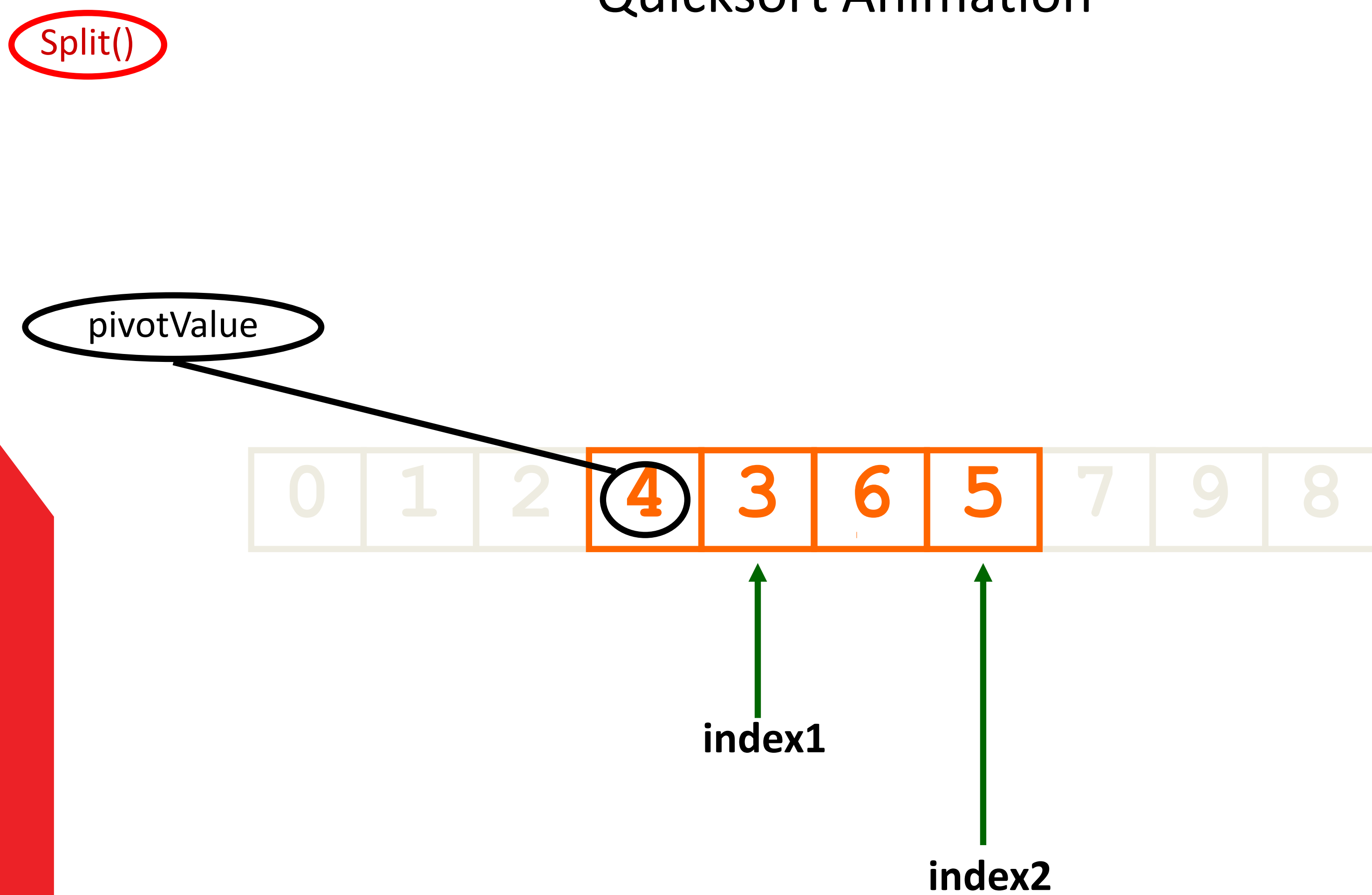


Quicksort the section below the pivotIndex

Quicksort Animation

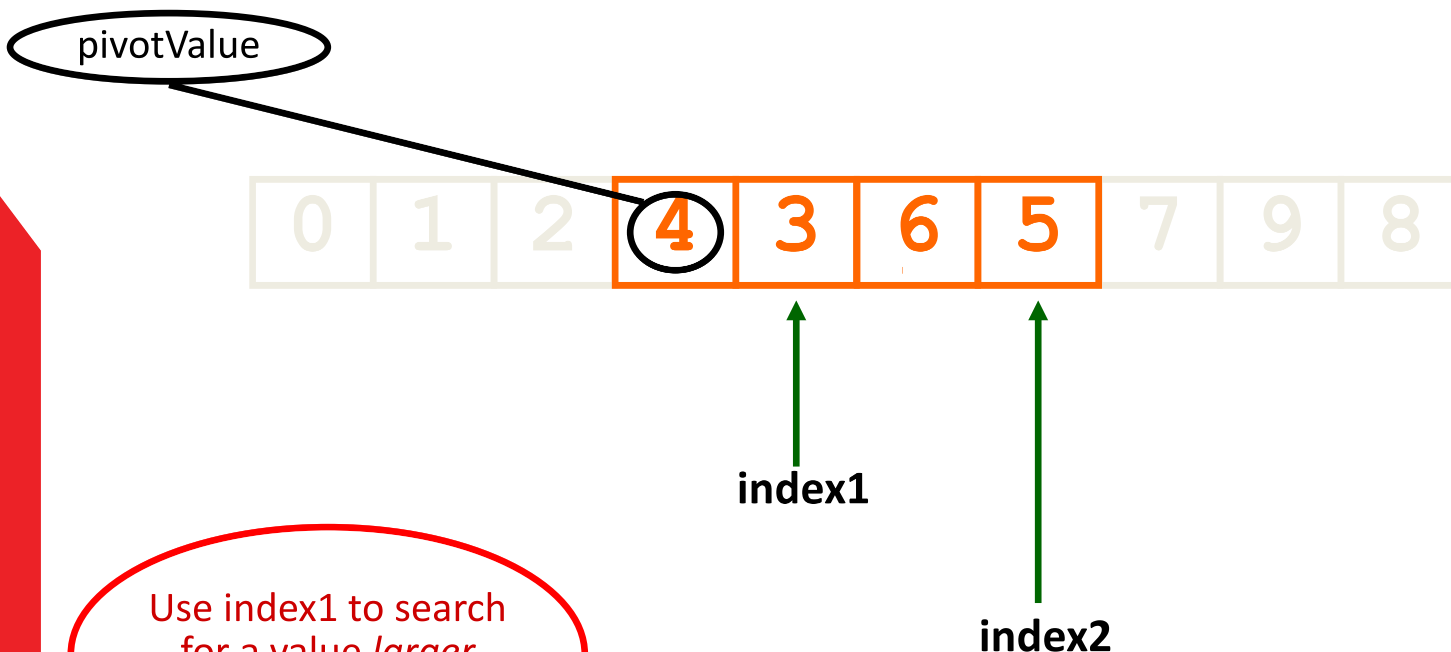


Quicksort Animation



Quicksort Animation

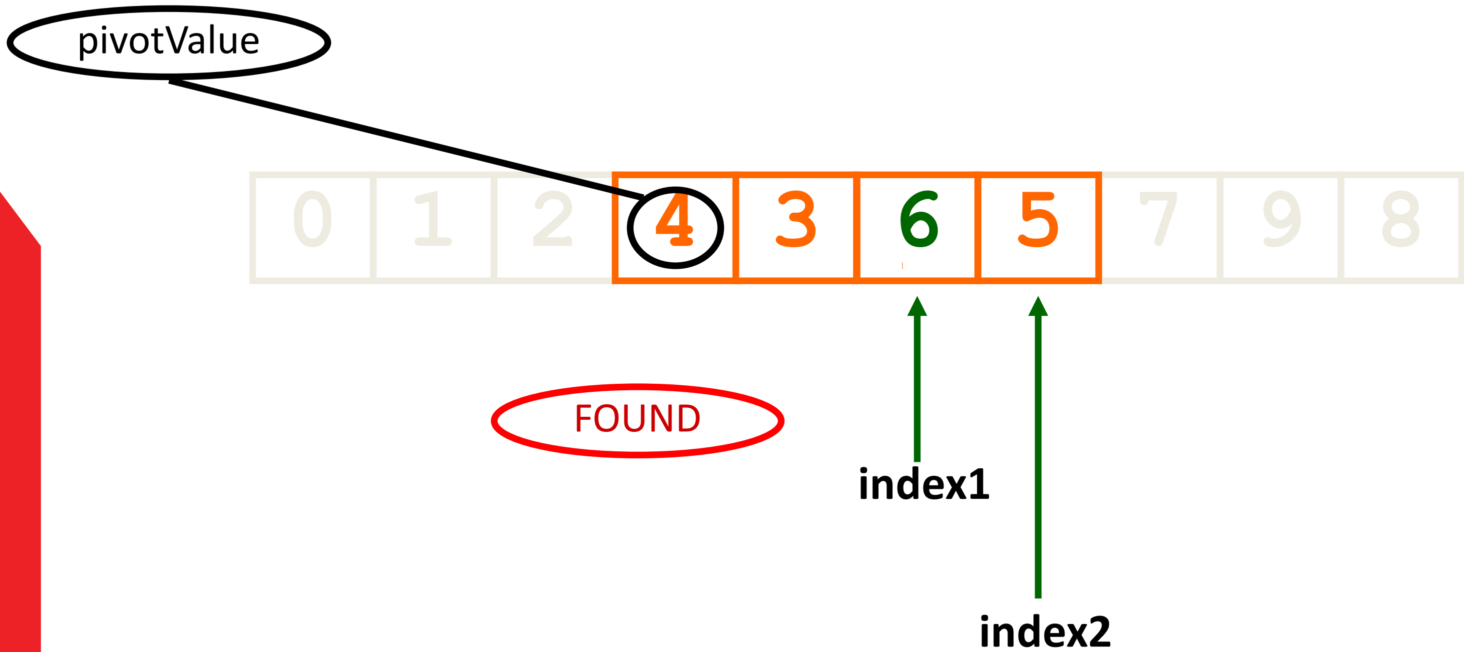
Split()



Use index1 to search for a value *larger* than the pivotValue

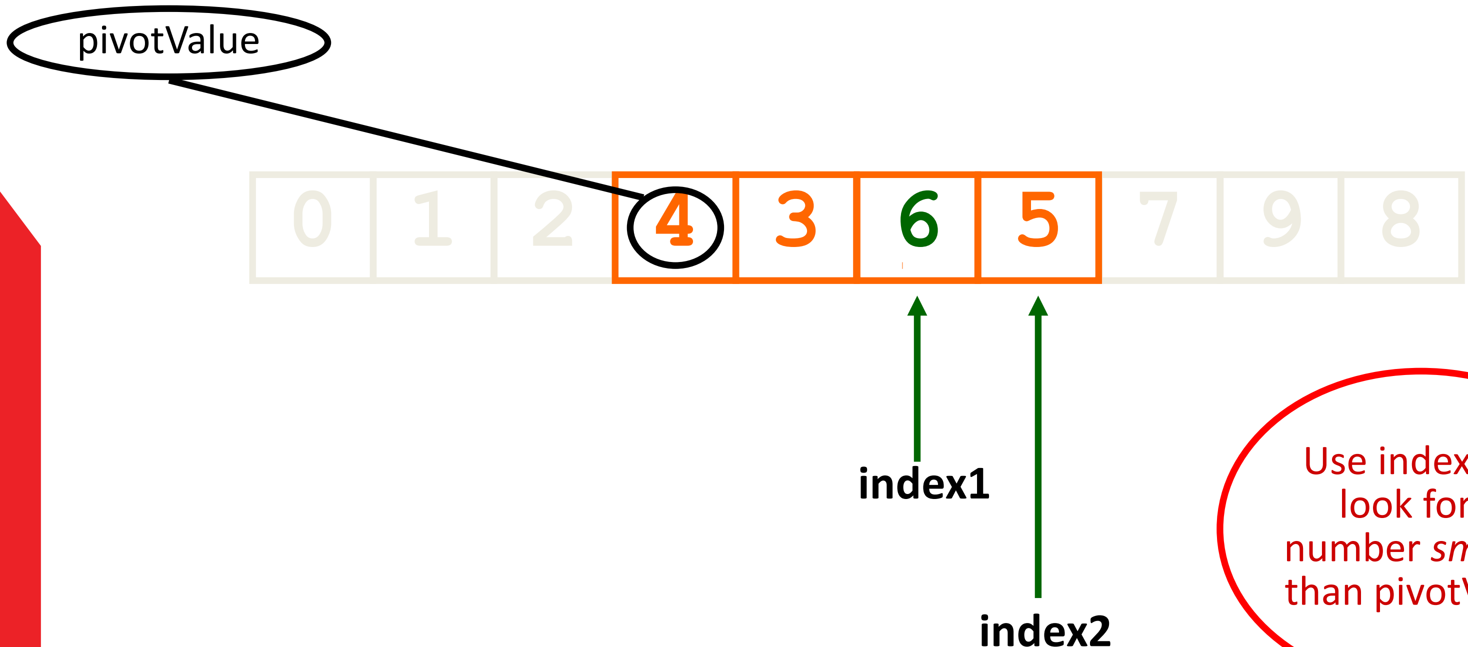
Quicksort Animation

Split()



Quicksort Animation

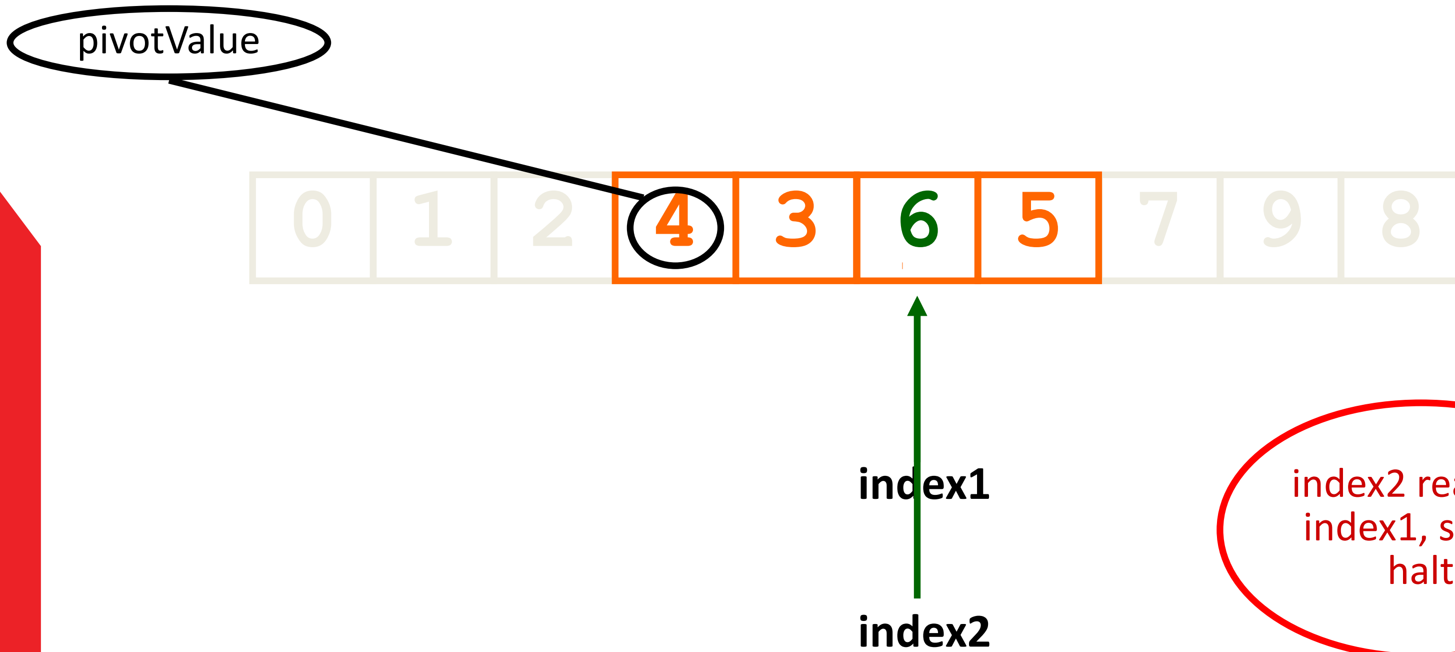
Split()



Use index2 to look for a number *smaller* than pivotValue

Quicksort Animation

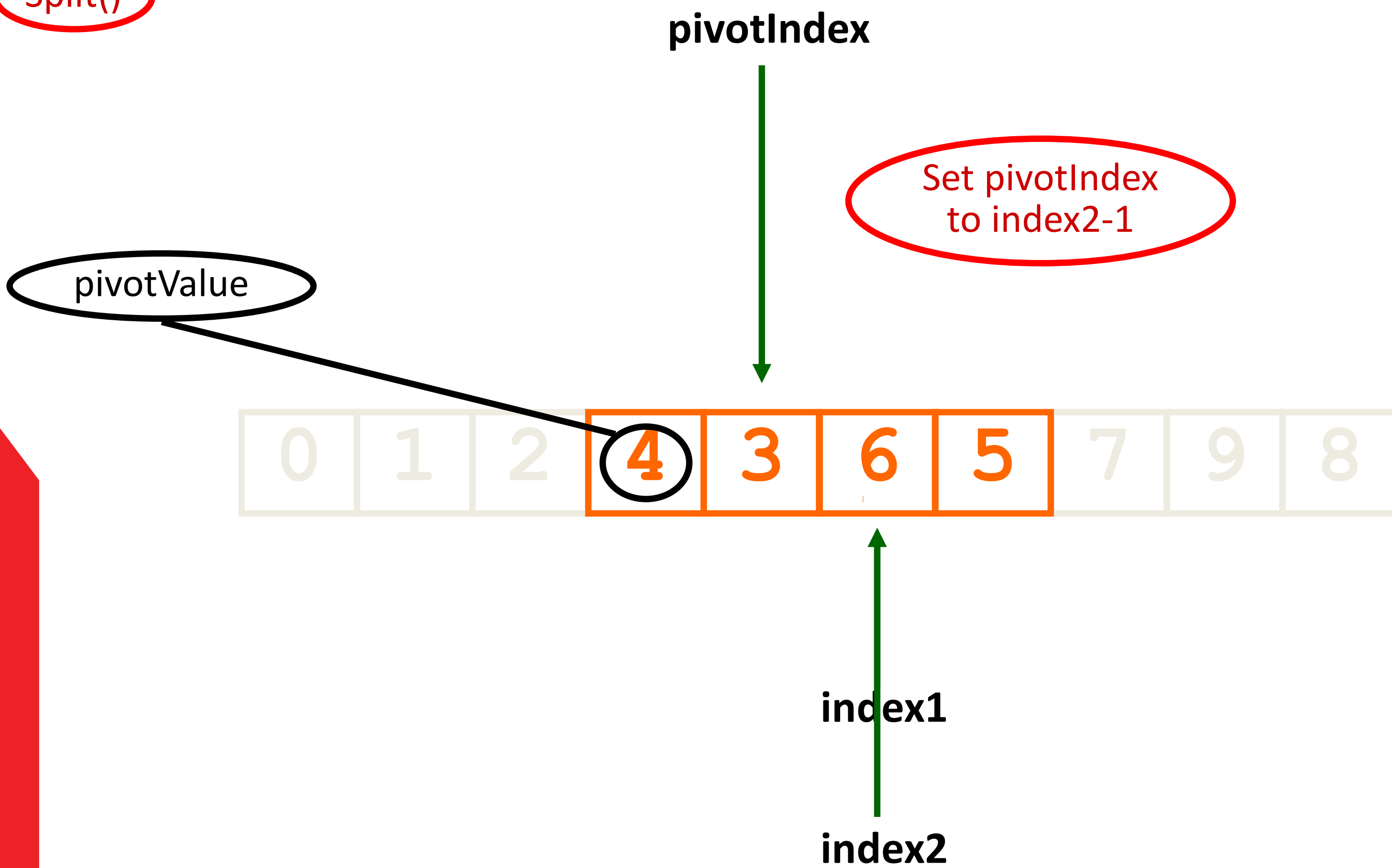
Split()



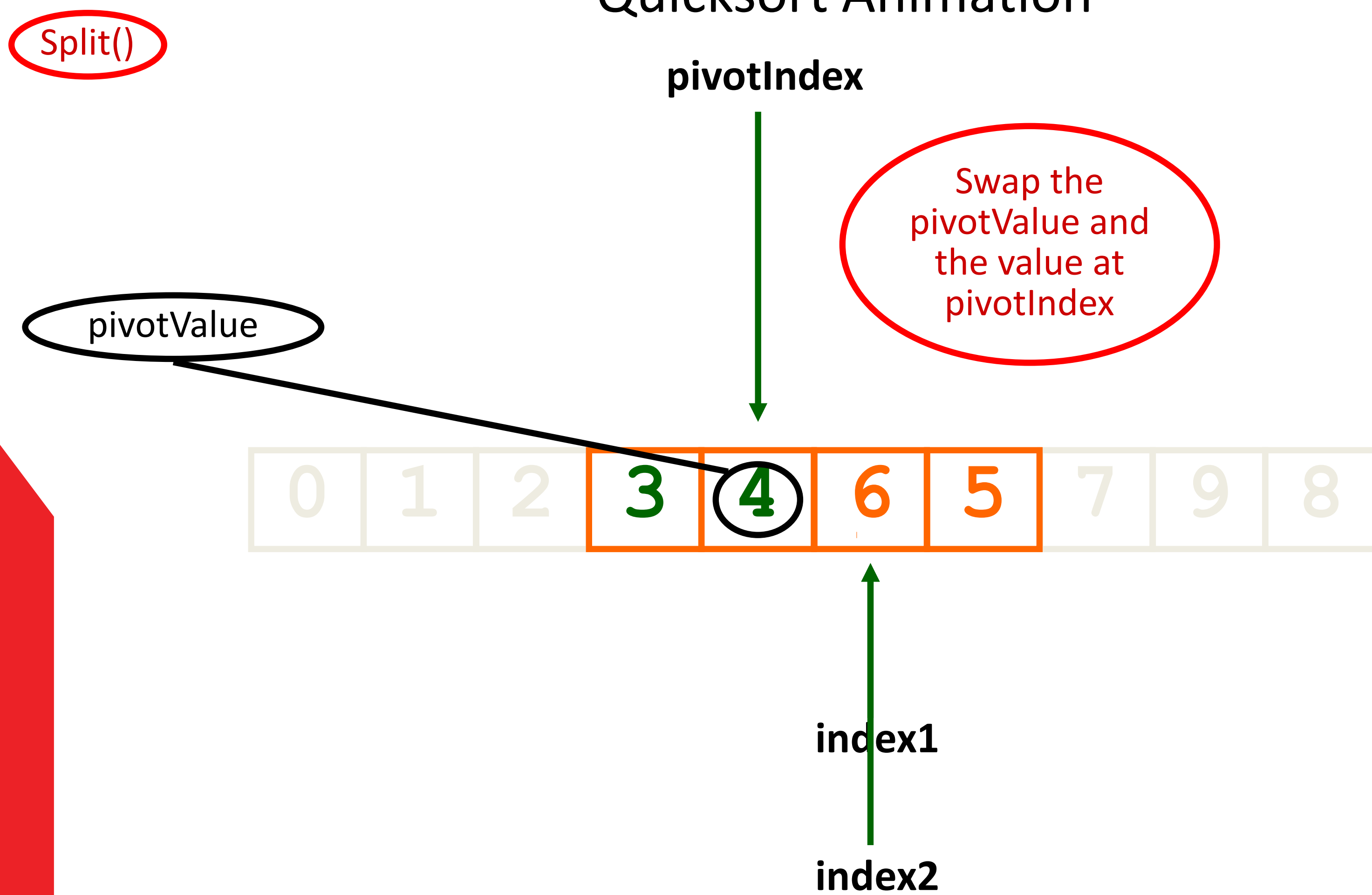
index2 reaches index1, so we halt

Quicksort Animation

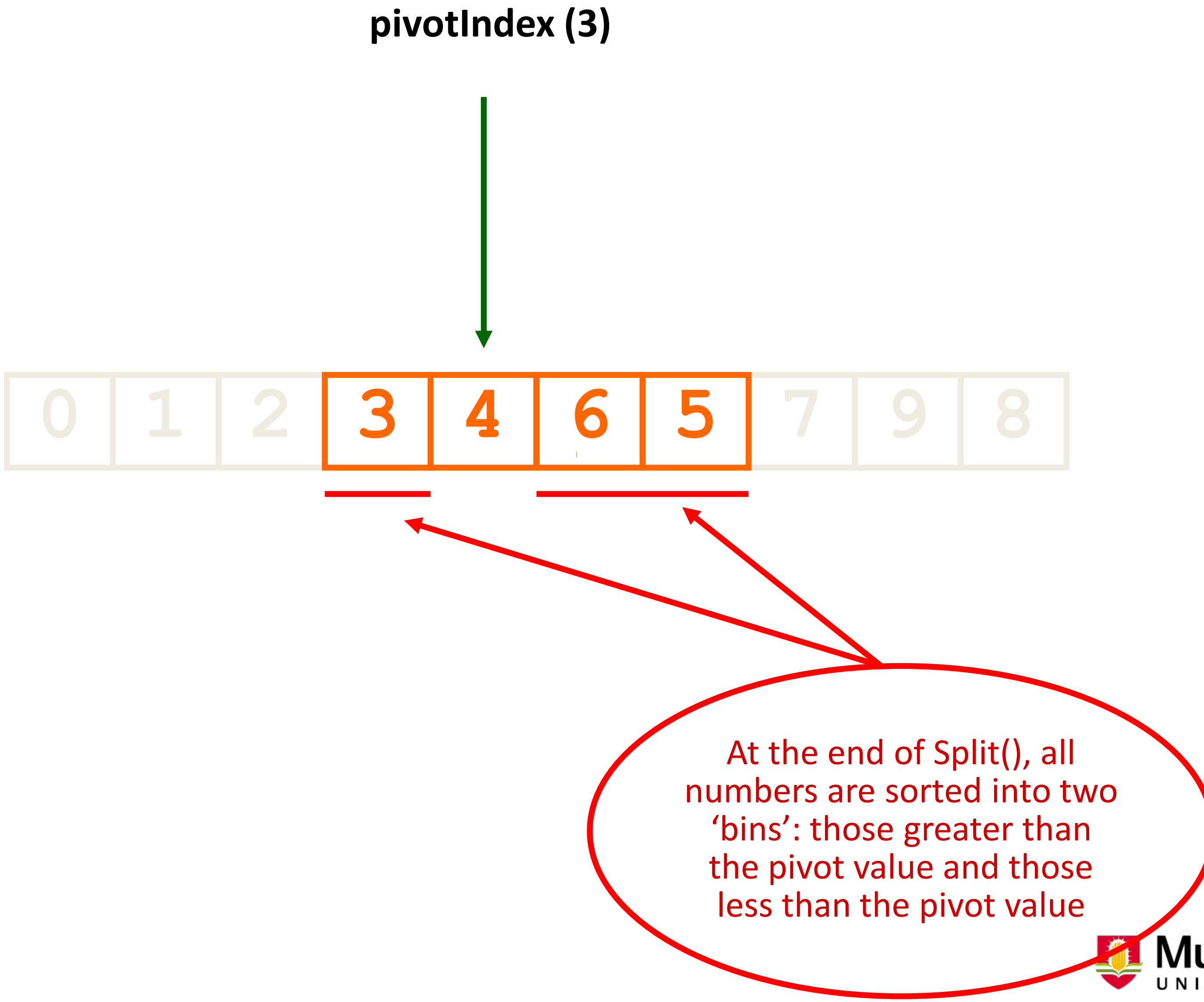
Split()



Quicksort Animation



Quicksort Animation



Murdoch UNIVERSITY

Quicksort Animation

pivotIndex (3)

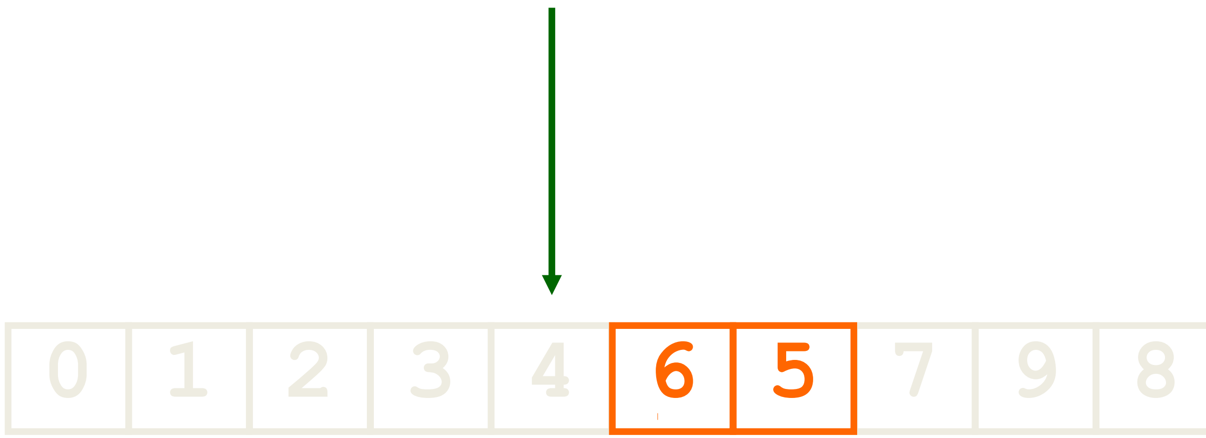
0	1	2	3	4	6	5	7	9	8

Only 1 value
below pivotIndex,
so do nothing

Murdoch
UNIVERSITY

Quicksort Animation

pivotIndex (3)



Only two values above pivotIndex, but out of order

Quicksort Animation

pivotIndex (3)

0	1	2	3	4	5	6	7	9	8

So swap them

Quicksort Animation

pivotIndex (2)

Only two values above pivotIndex, but out of order

Quicksort Animation

pivotIndex (2)

0	1	2	3	4	5	6	7	8	9

So swap them

Quicksort Animation

0	1	2	3	4	5	6	7	8	9

Phew!!

Done!!

Readings

- Textbook Chapter Searching and sorting Algorithms. Diagrams in the textbook also explain step by step.

- Reference book, Introduction to Algorithms. For further study, see part of the book called Sorting and Order Statistics. It contains a number of chapters on sorting.

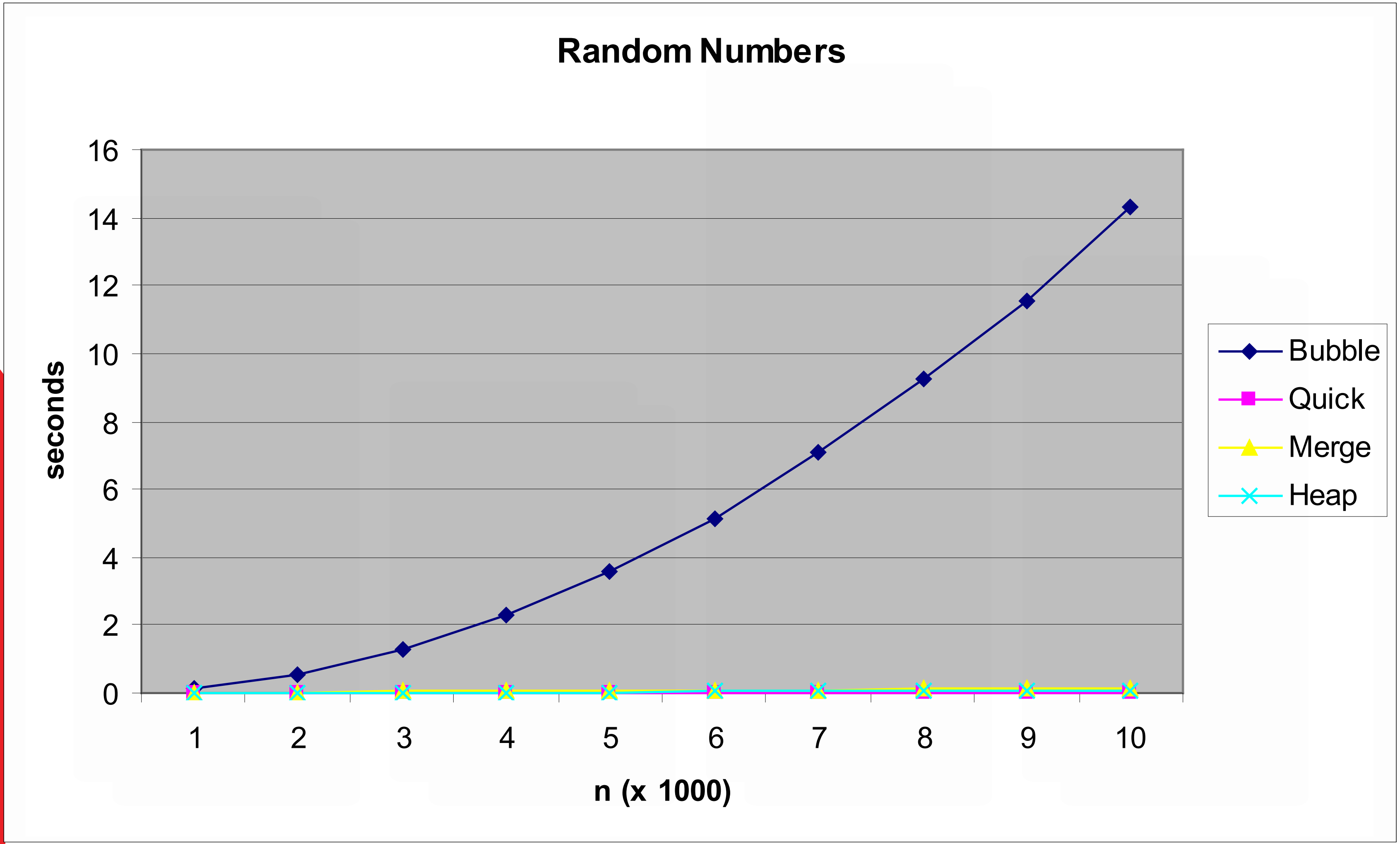
Murdoch
UNIVERSITY

Data Structures and Abstractions

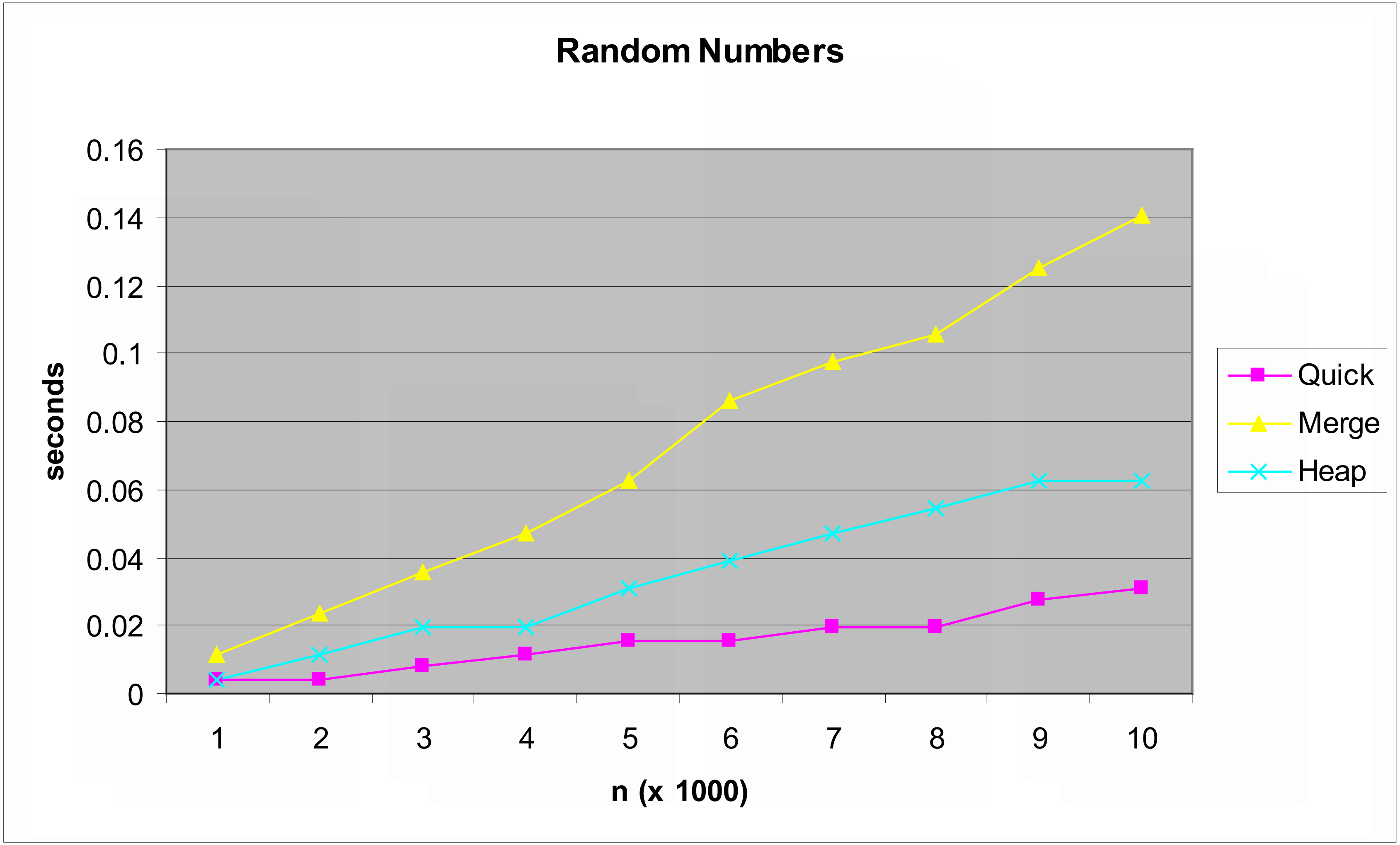
Empirical Comparisons, and the STL Sorts

Lecture 27

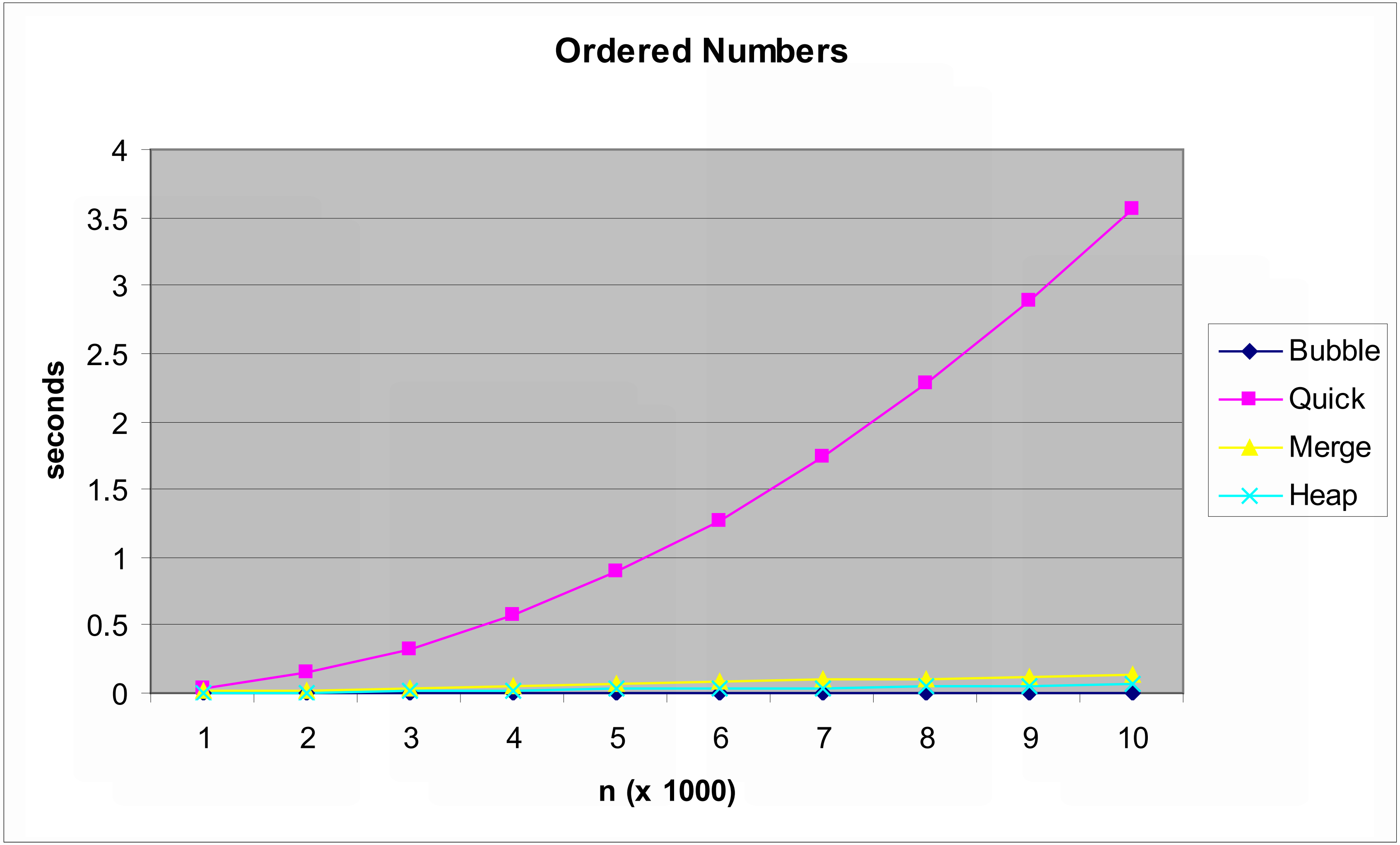
Empirical Comparison 1[1]



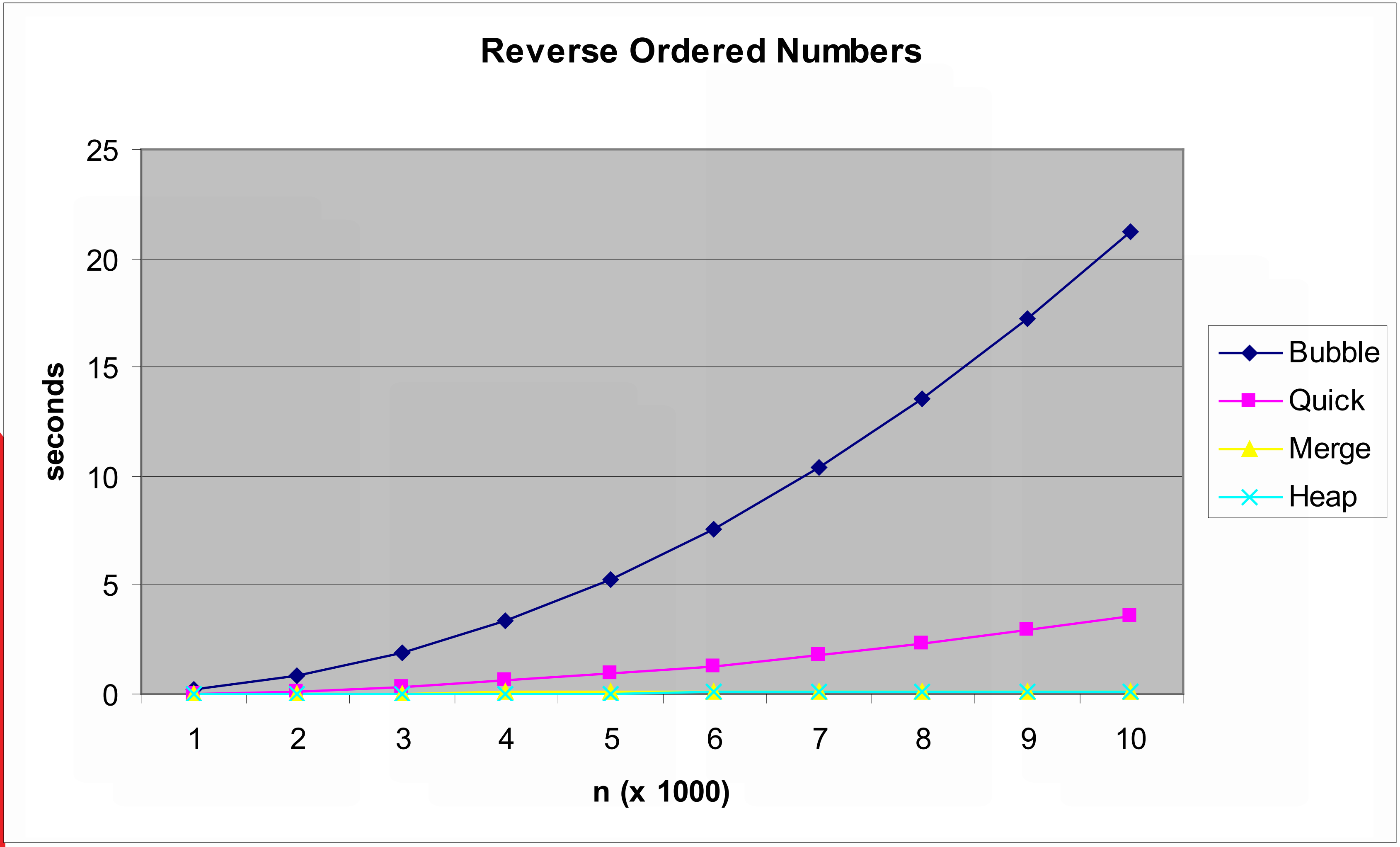
Empirical Comparison 2



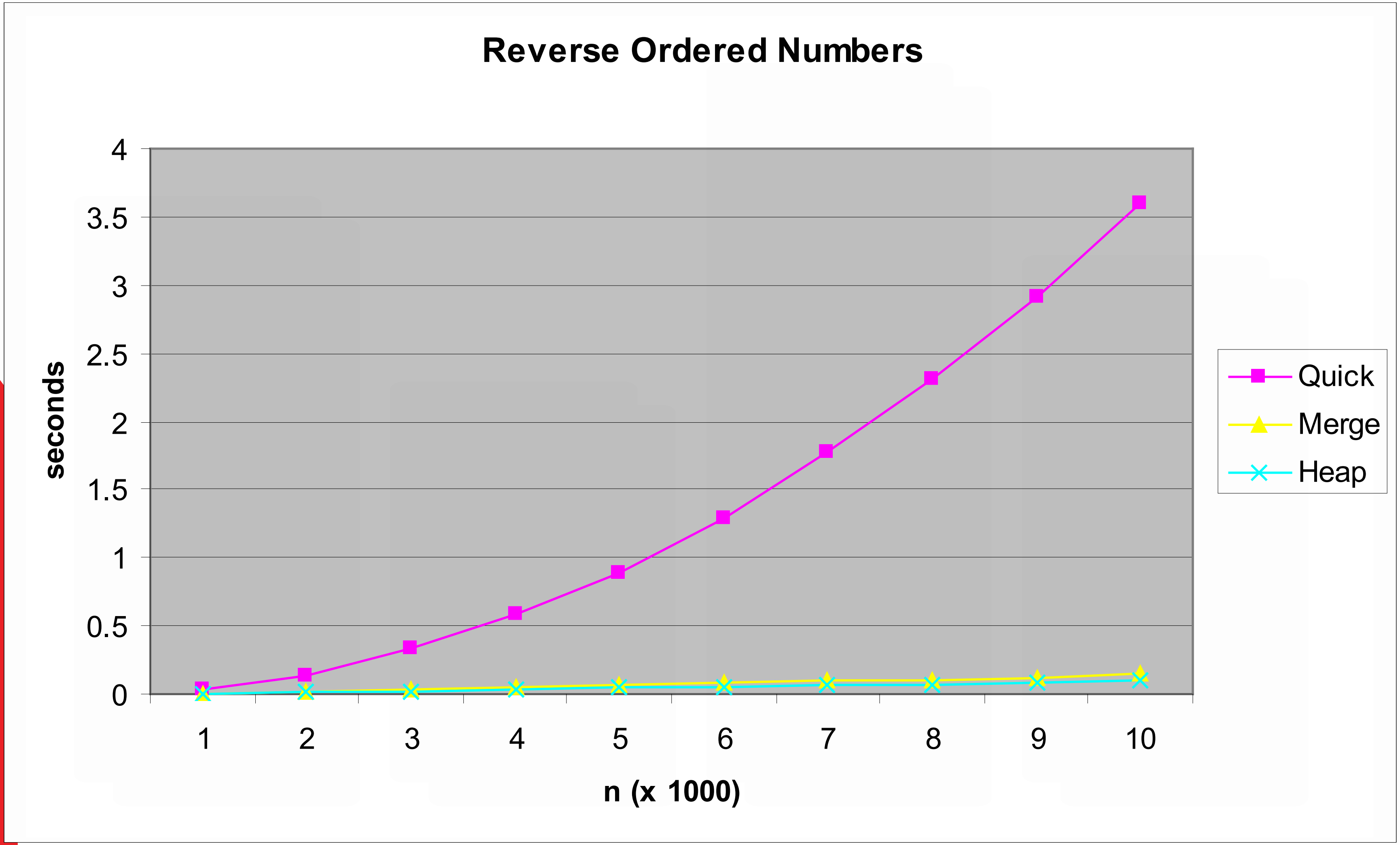
Empirical Comparison 3



Empirical Comparison 4



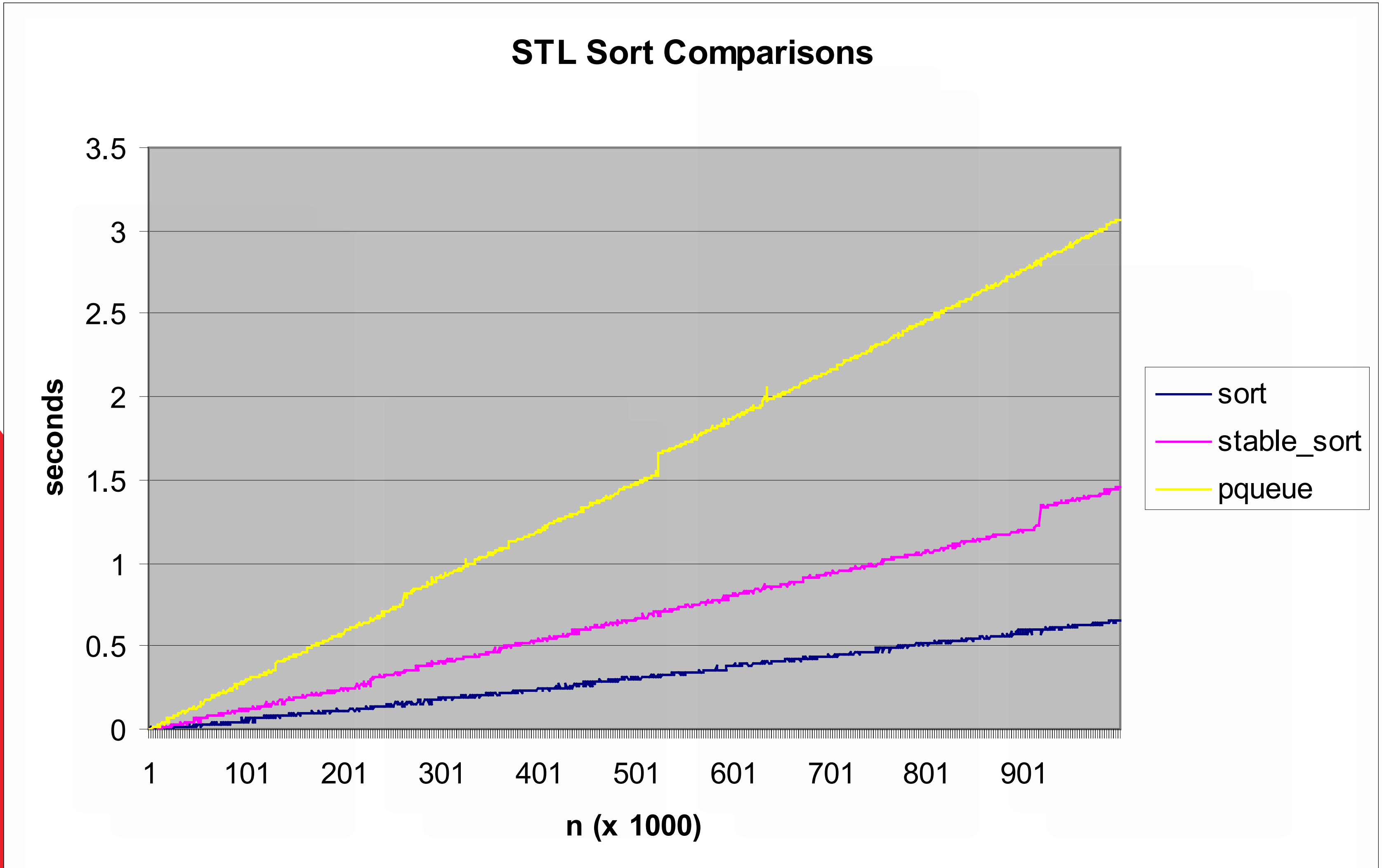
Empirical Comparison 5



STL Sorts

- There are a number of sort routines available in the STL algorithm library, a priority queue, which acts as a heap sort, plus some templates that have sorts of their own.
- **`sort(thing.begin(), thing.end())`** is a quicksort algorithm.
- **`stable_sort(thing.begin(), thing.end())`** does a stable sort, but I could not find definitive information on the algorithm used. However it is described as being like the sort algorithm, in which case, the type of split or partitioning routine will determine stability. But see Silicon Graphics site [1] notes where it is made explicit that stable_sort uses merge sort.
- **`pqueue<something>`** is a heap: put data into it and then pull it out and it is in order. [2]
- These are all very, very fast indeed: much faster than the ones any particular individual can write.
- This is because they have been written, reviewed, optimised etc. by multiple experts.

Empirical Comparison 6



Less Than Operator

- Note that these sorts require that a less than operator (<) be available for the ‘things’ being sorted.
- Therefore, if you are sorting your own objects, you must overload a less than operator within the class to which they belong.
- To overload a < operator for a Circle class: [1]

```
bool Circle::operator < (const Circle &other)
{
    return (m_radius < other.m_radius);
}
```

Murdoch UNIVERSITY

Readings

- Textbook, chapter Standard Template Library, section on Algorithms.

Murdoch University

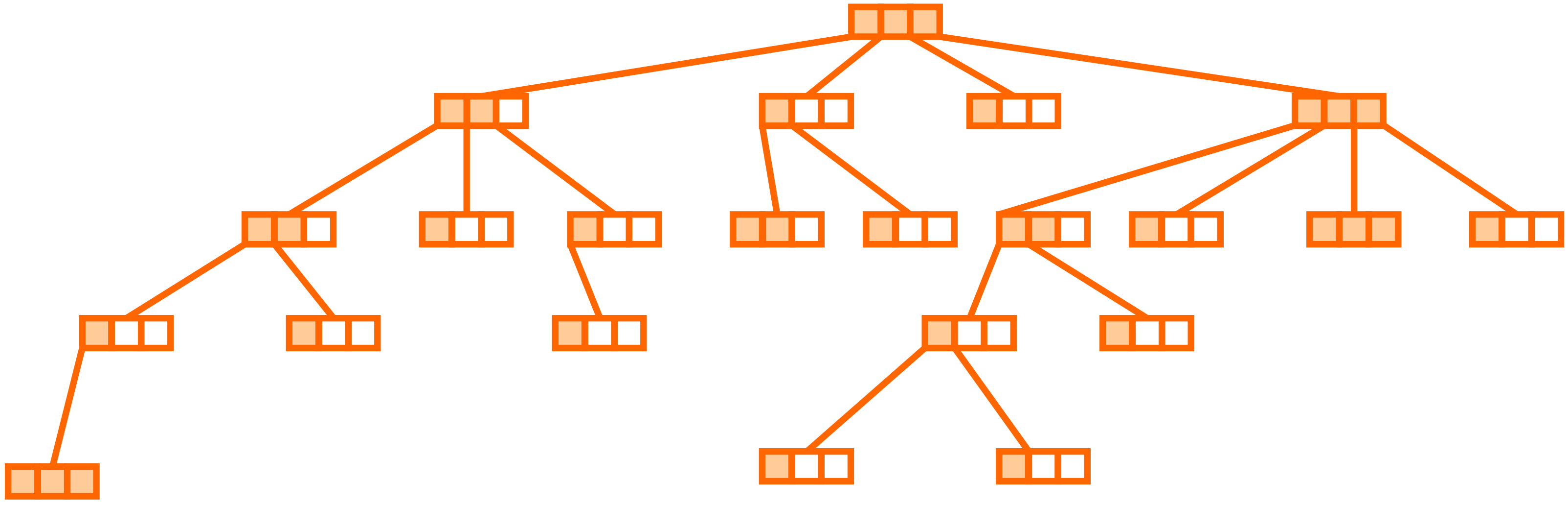
Data Structures and Abstractions

Trees and Tree Searching

Lecture 28

Trees

- Trees are ADS where every Node has directional links to one or more nodes underneath it. [1]
- An m-tree is a node with 0:*m* links and 1:*m*-1 pieces of data in each node. For example a 4-tree (quadtree or 4-way tree) might look something like this:



Tree Definitions

- The top node is called the root. [1]
- Any node that is not the root node is a child of some parent.
- Any node that has one or more children is a parent.
- Nodes connected to same parent are called siblings. [2]
- Any node that has no children is called a leaf.
- Any part of the tree smaller than the whole is called a subtree.

Tree Use [1]

- The back-ends of databases.
- Data stores that are not databases.
- Problem solving.
- Game playing.
- Graphics and virtual reality: for tracking line of sight as well as storing screen objects.
- Graph theory (e.g. path finding).

The algorithm used to insert data into the tree will vary from application to application.

Traversal

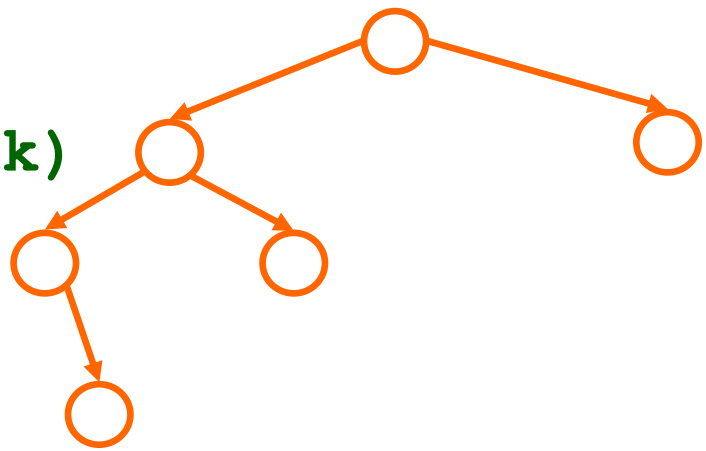
- Traversing a tree involves going to every node.
- This needs to be done for processes such as printing, gathering statistics, end-of-month calculations, searching etc.
- It can be done either in-order, pre-order or post-order.
- The method chosen depends on the application.
- **In the traversal examples, we look at a 2-way or *binary* tree**, as this is the simplest to understand.

Murdoch UNIVERSITY

In-Order Traversal

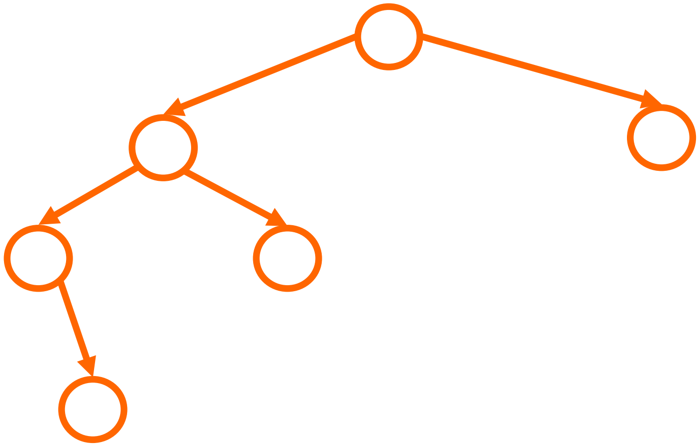
Examples don't have the terminating condition for recursion – see note [1]

```
ProcessNode (node) [1]
    ProcessNode (leftLink)
    Process this node
    ProcessNode (rightLink)
End ProcessNode
```



In-Order Traversal Animation

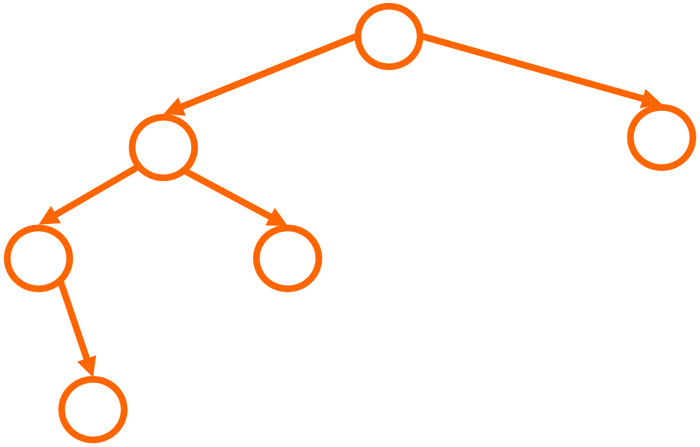
```
ProcessNode (node)
    ProcessNode (leftLink)
    Process this node
    ProcessNode (rightLink)
End ProcessNode
```



END

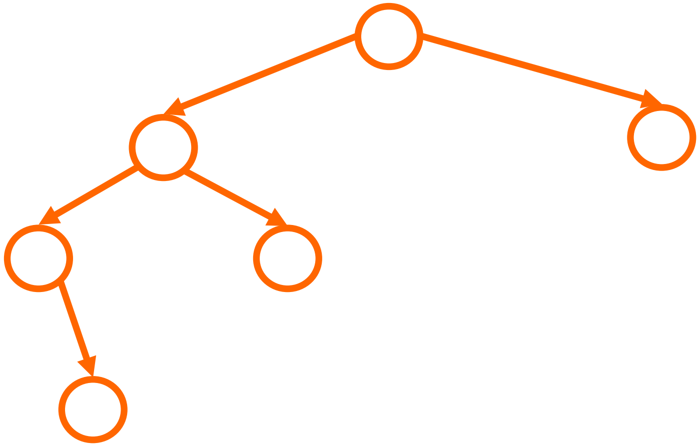
Pre-Order Traversal

```
ProcessNode (node)
   Process this node
   ProcessNode (leftLink)
   ProcessNode (rightLink)
End ProcessNode
```



Pre-Order Traversal Animation

```
ProcessNode (node)
   Process this node
   ProcessNode (leftLink)
   ProcessNode (rightLink)
End ProcessNode
```

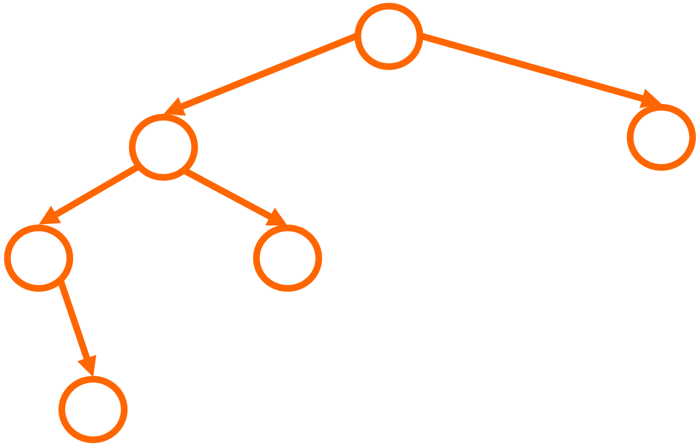


Murdoch UNIVERSITY

END

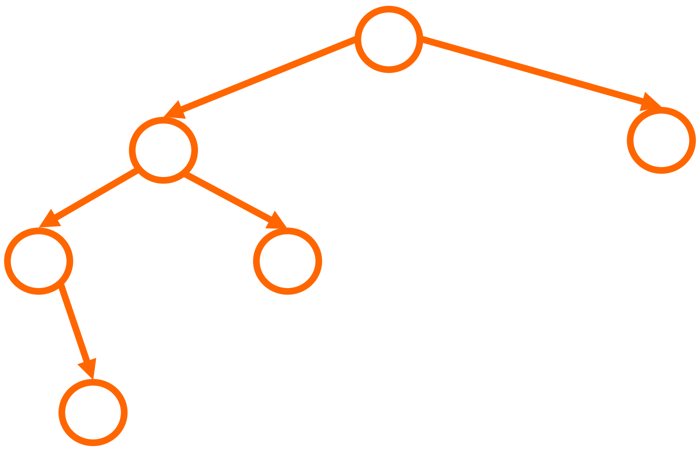
Post-Order Traversal

```
ProcessNode (node)
    ProcessNode (leftLink)
    ProcessNode (rightLink)
    Process this node
End ProcessNode
```



Post-Order Traversal Animation

```
ProcessNode (node)
    ProcessNode (leftLink)
    ProcessNode (rightLink)
    Process this node
End ProcessNode
```



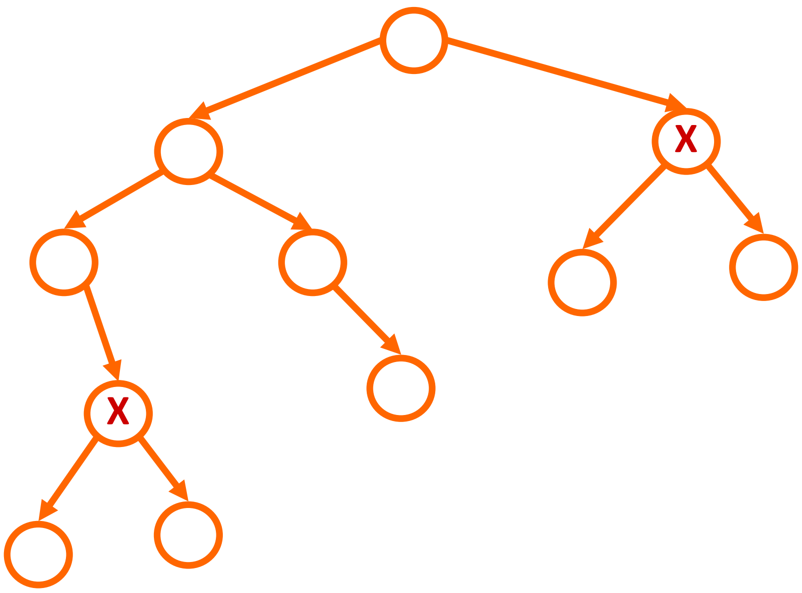
END

Tree Searching

- Trees will also need to be searched, and there are many different search algorithms available.
- When not using heuristics, there are two main ways in which to do a logical search:
  - Depth first
    - which is simply a search done in pre-order stopping when the target data is found;
    - the aim is to find ***any*** match to the target;
    - this is commonly used when trying to find the one unique match.
  - Breadth first
    - where the nodes are searched in layers, down from the top;
    - this search aims to find the match that is ***closest*** to the start;
    - a common use for this is in game playing: you want to win as soon as possible.

Depth First Search Animation

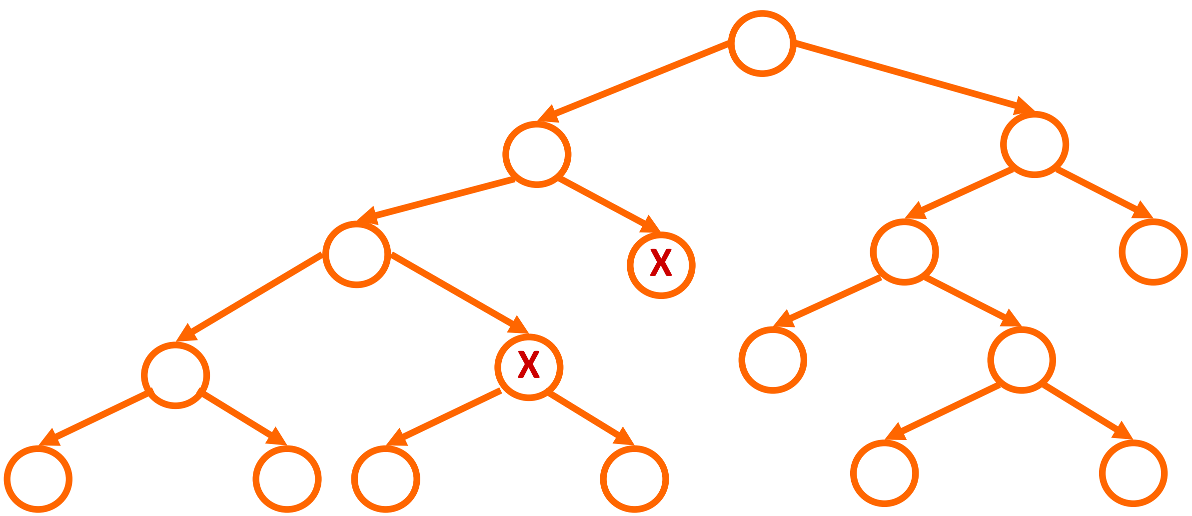
```
Search (node) : boolean
    boolean found
    found = target at this node
    IF not found
        found = Search (leftLink)
        IF not found
            found = Search (rightLink)
        ENDIF
    ENDIF
    return found
End Search
```



END

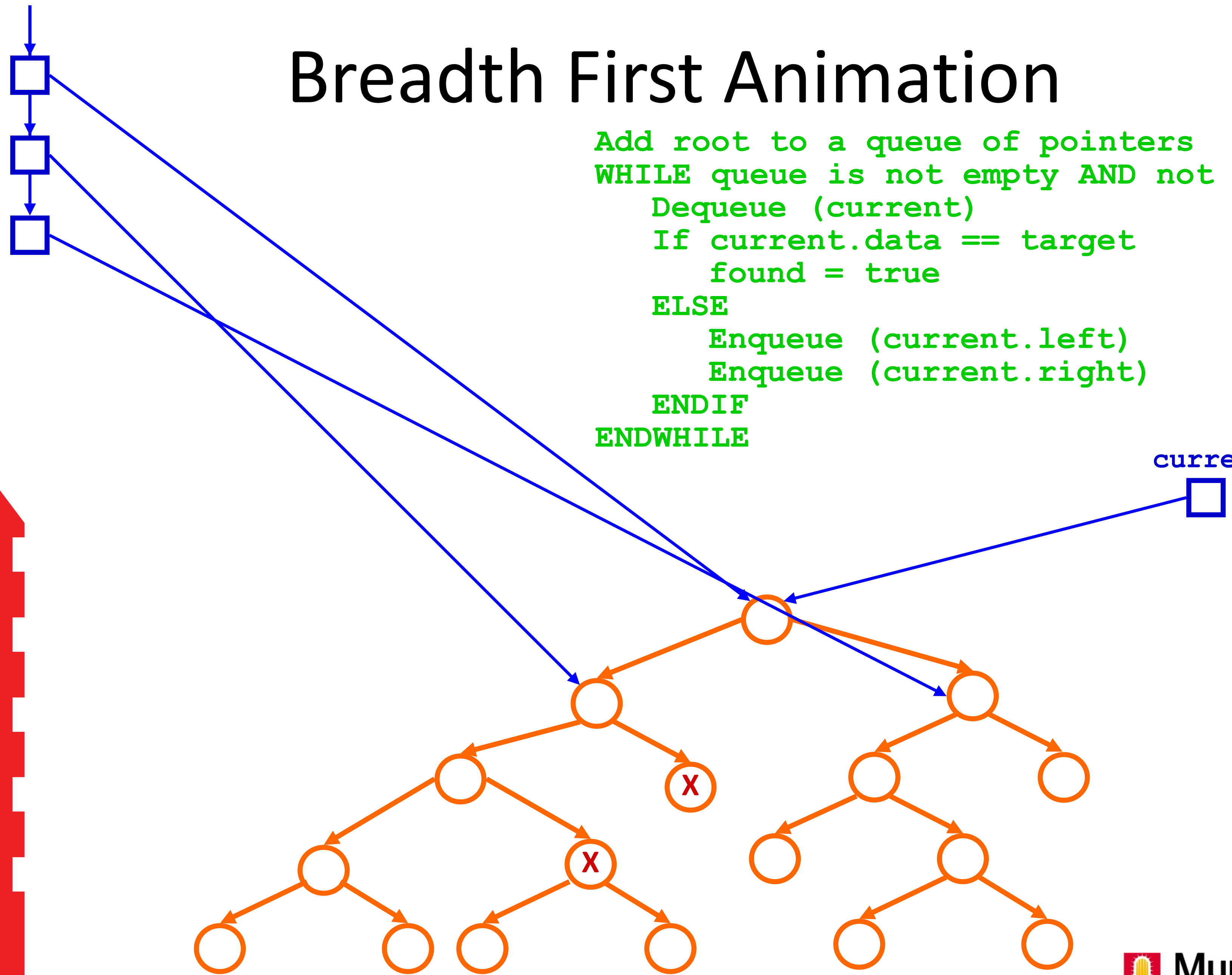
Breadth First Animation [1]

```
Add root to a queue of pointers
WHILE queue is not empty AND not found
   Dequeue (current)
   If current.data == target
      found = true
   ELSE
      Enqueue (current.left)
      Enqueue (current.right)
   ENDIF
ENDWHILE
```



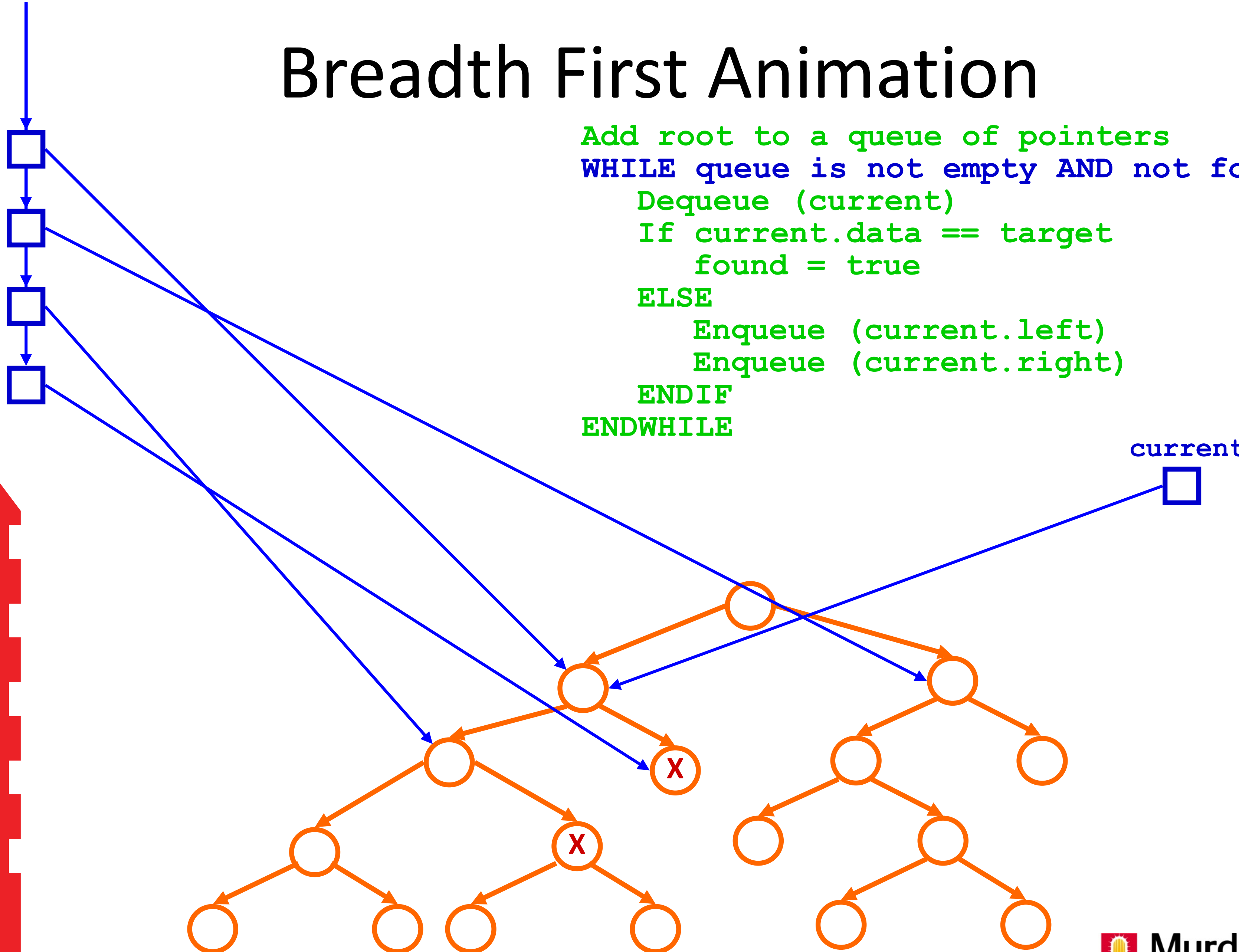
Breadth First Animation

```
Add root to a queue of pointers
WHILE queue is not empty AND not found
   Dequeue (current)
   If current.data == target
      found = true
   ELSE
      Enqueue (current.left)
      Enqueue (current.right)
   ENDIF
ENDWHILE
```



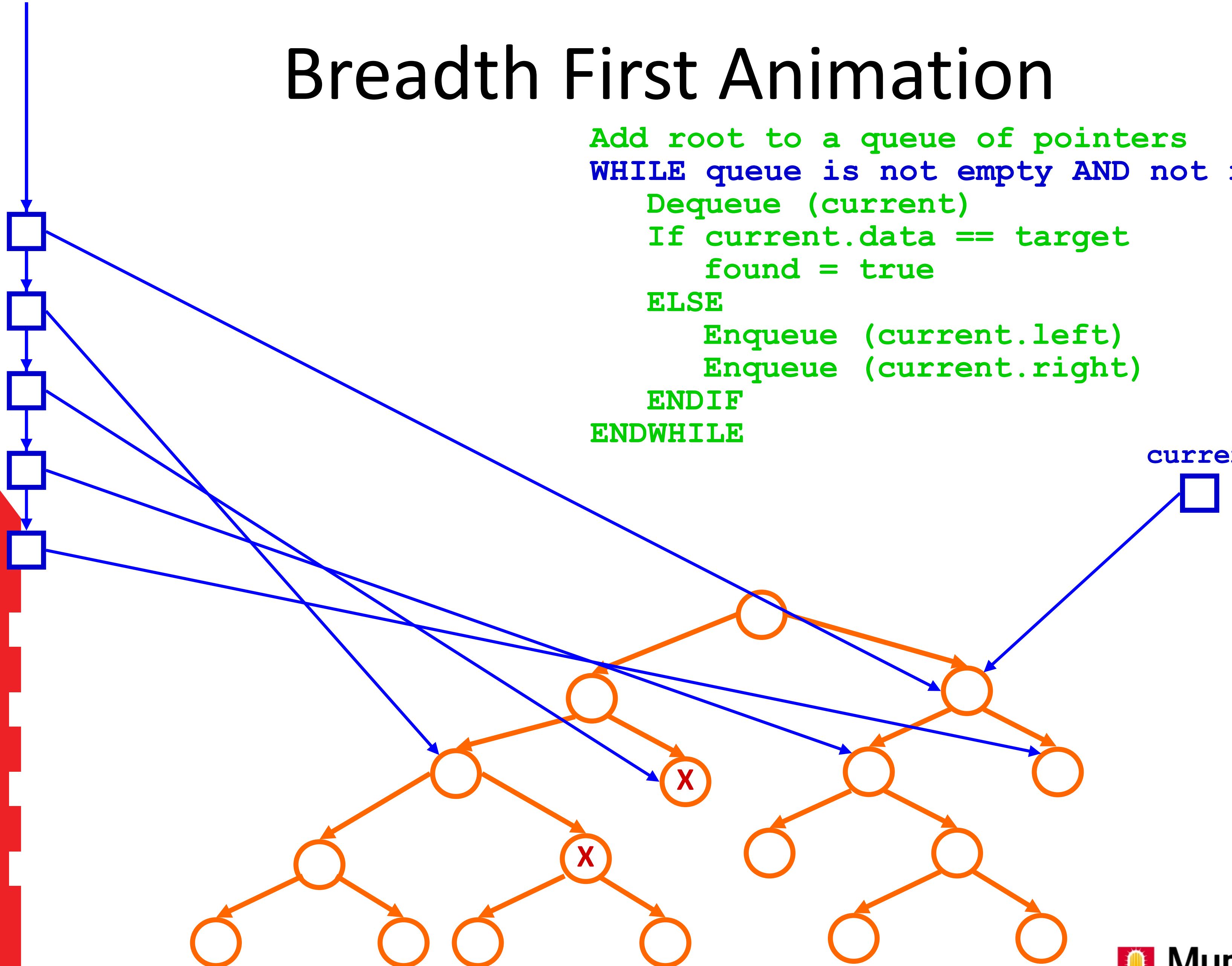
Breadth First Animation

```
Add root to a queue of pointers
WHILE queue is not empty AND not found
    Dequeue (current)
    If current.data == target
        found = true
    ELSE
        Enqueue (current.left)
        Enqueue (current.right)
    ENDIF
ENDWHILE
```



Breadth First Animation

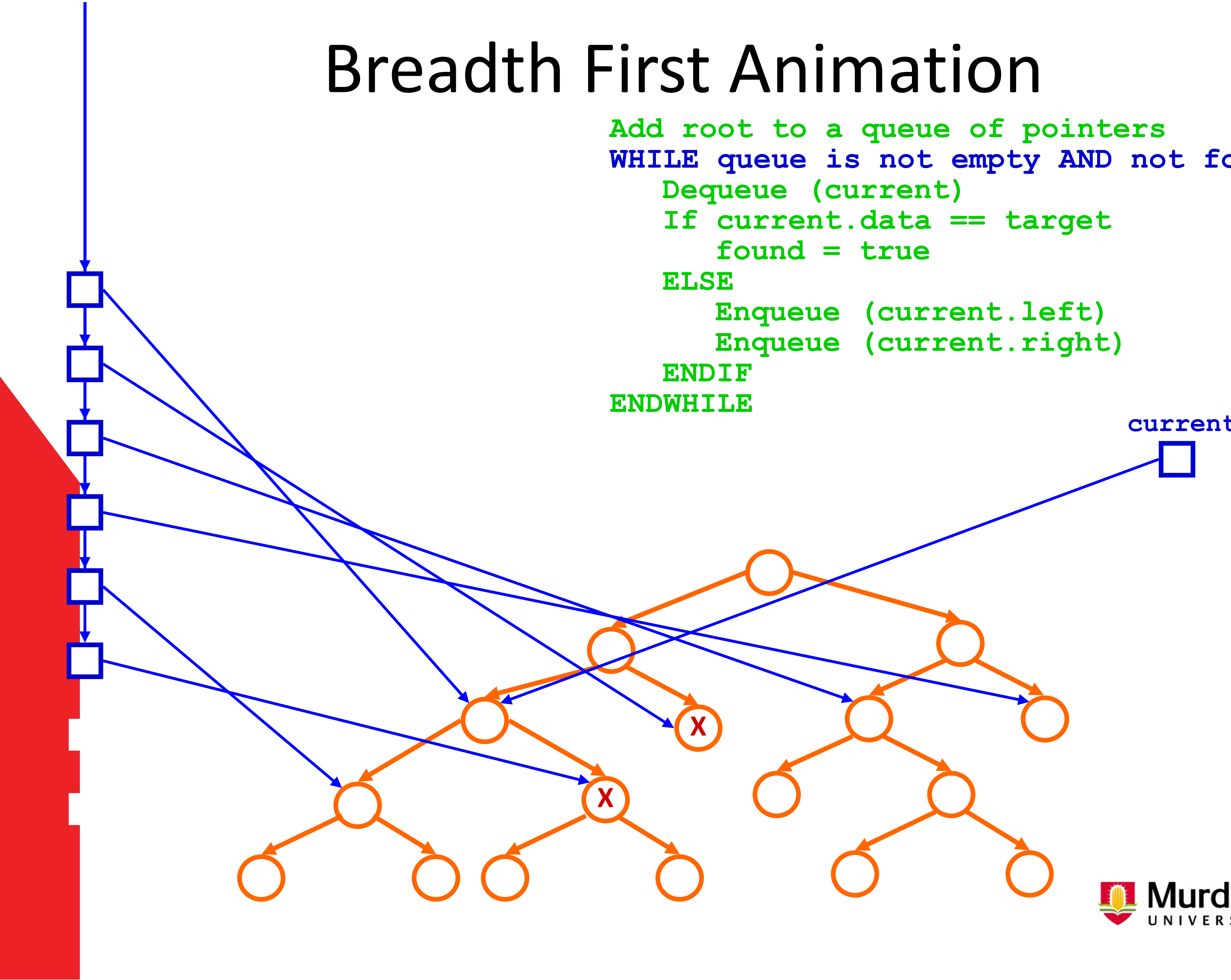
```
Add root to a queue of pointers
WHILE queue is not empty AND not found
   Dequeue (current)
   If current.data == target
      found = true
   ELSE
      Enqueue (current.left)
      Enqueue (current.right)
   ENDIF
ENDWHILE
```



Murdoch UNIVERSITY

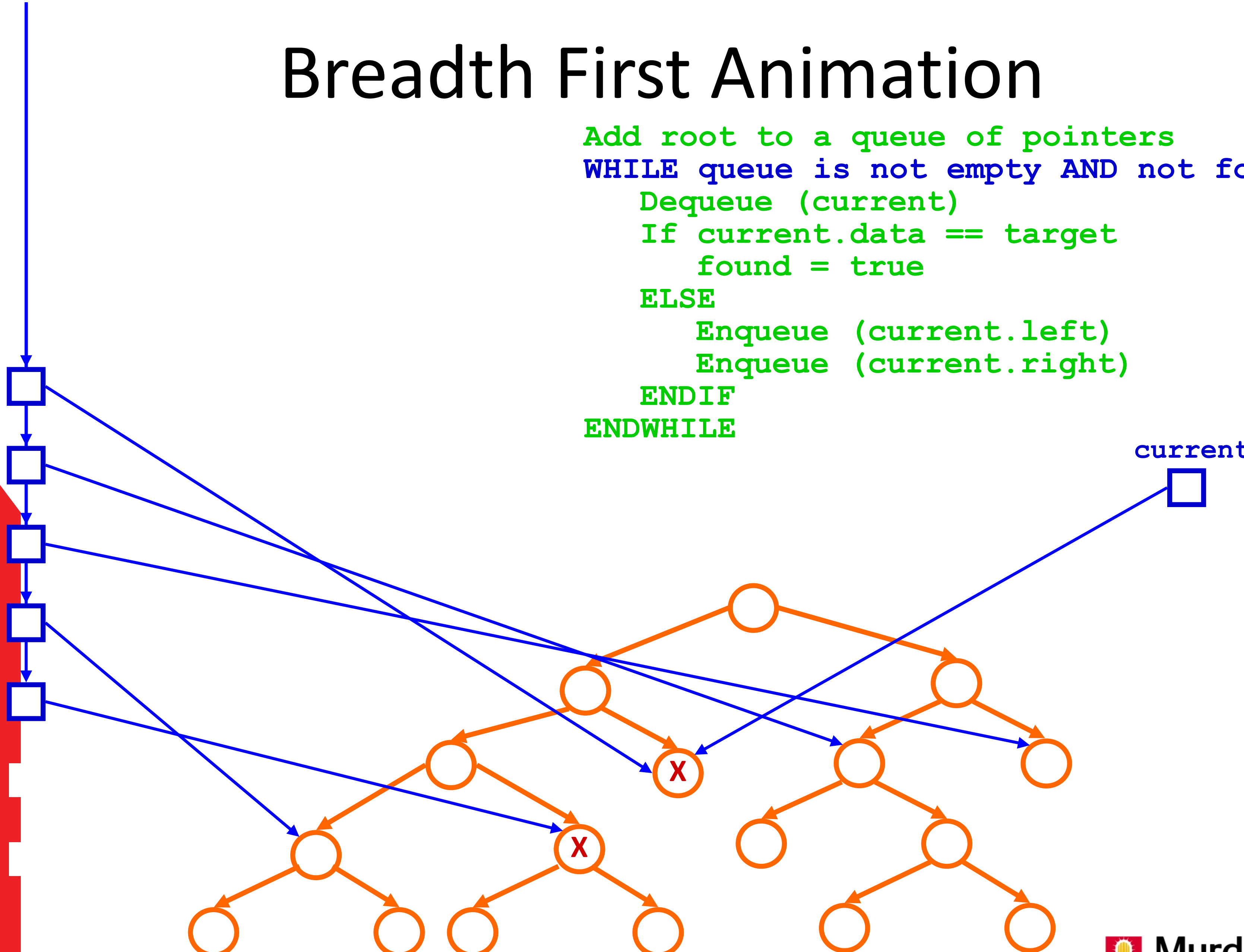
Breadth First Animation

```
Add root to a queue of pointers
WHILE queue is not empty AND not found
   Dequeue (current)
   If current.data == target
      found = true
   ELSE
      Enqueue (current.left)
      Enqueue (current.right)
   ENDIF
ENDWHILE
```



Breadth First Animation

```
Add root to a queue of pointers
WHILE queue is not empty AND not found
    Dequeue (current)
    If current.data == target
        found = true
    ELSE
        Enqueue (current.left)
        Enqueue (current.right)
    ENDIF
ENDWHILE
```



Murdoch University

END

Readings

- Textbook: Chapter on Binary Trees
  - Should go through the programming example at the end of the chapter.
- Textbook: Chapter on Recursion
  - Revise the concept covered in earlier units and be able to implement recursive routines.
  - Recursion vs Iteration
- Further exploration:
  - Reference book, Introduction to Algorithms. For further study, there are many tree and tree algorithms described in the reference book. For this unit, the lecture notes, practical work and the textbook is sufficient.

Murdoch UNIVERSITY

Murdoch UNIVERSITY

Data Structures and Abstractions

Binary Search Trees

Lecture 29

Introduction to ADS Sorted Data Stores

- As pointed out in the earlier lecture, trees are used for problem solving, game playing, virtual reality and data storage, amongst other things.
- When used for data storage they are always built so that the data is sorted as it is inserted.
- We will be looking at several different sorted trees including Binary Search Trees, AVL Trees, Multiway Trees, B-Trees and B+ trees. [1]
- In later lectures we will also consider non-sorted trees used to store information during graph processing.

Murdoch UNIVERSITY

The Data to be Stored

- The data stored in the tree can either be the actual data or a pointer/index to the actual data.
- The actual **data** stored will almost always contain a ***key*** plus other data.
- The key is used to place (order) the data in the container.
- Examples of keys are account numbers, membership numbers, names, or keys calculated from some part of the data.
- The key should be unique to enable the BST to be more efficient.
- It also possible to have secondary keys, where a list, array or tree is ‘overlayed’ on the first structure giving a different sorted order.

Murdoch UNIVERSITY

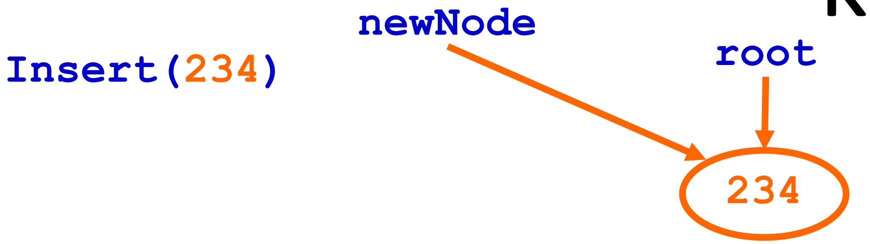
Binary Search Trees

- Binary trees are trees where
  - every node has 1 piece of data and two pointers (left, right)
  - every node, except root, can have a parent pointer [1]
  - therefore every node has 0:2 children
- Binary search trees are binary trees where
  - every node has data that is greater than the data in all nodes to the left of it.
  - every node has data that is less than the data in all nodes to the right of it.
- Note that this contrasts with the heap (see later), where a node's data was always guaranteed to be less than (for a min-heap) or greater than (for a max-heap) all data in its subtree. [2]
- Since the data sorting is based on a unique key, there is normally no two identical sets of data. [3]

BST Algorithms

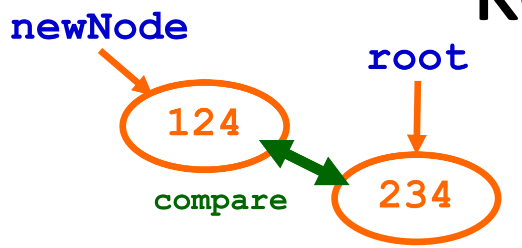
- Almost all BST algorithms are recursive as this makes them very simple.
- However, the root node might be treated differently because it has no parent but should it? [1]
- All the methods require that root has been set to NULL in the constructor.
- Traversal of a BST is almost always done either in-order or pre-order. [2]

BST Insert Animation (for integer key)



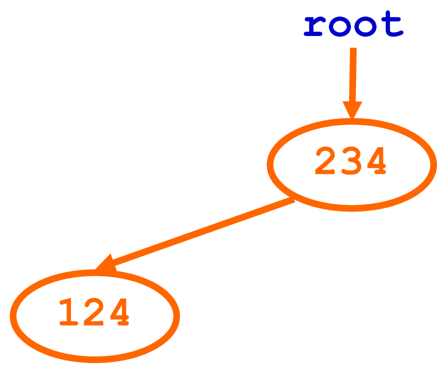
BST Insert Animation (for integer key)

`Insert(124)`

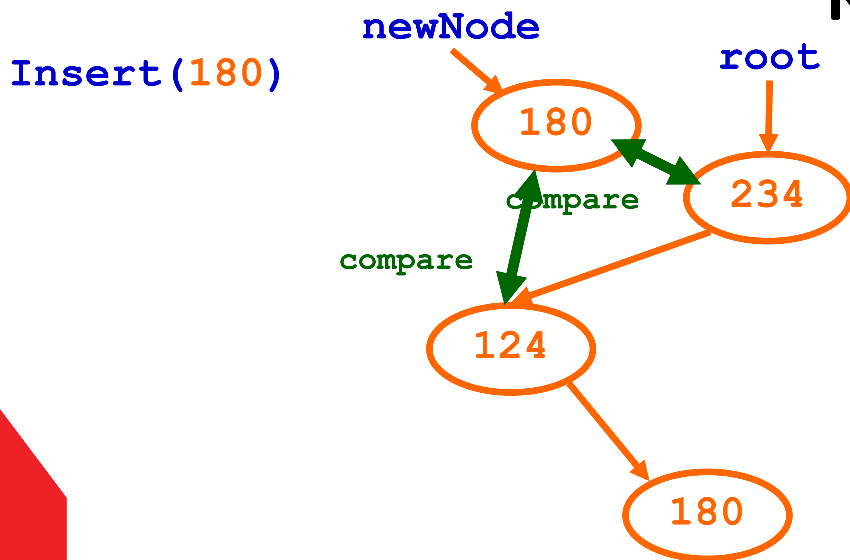


BST Insert Animation (for integer key)

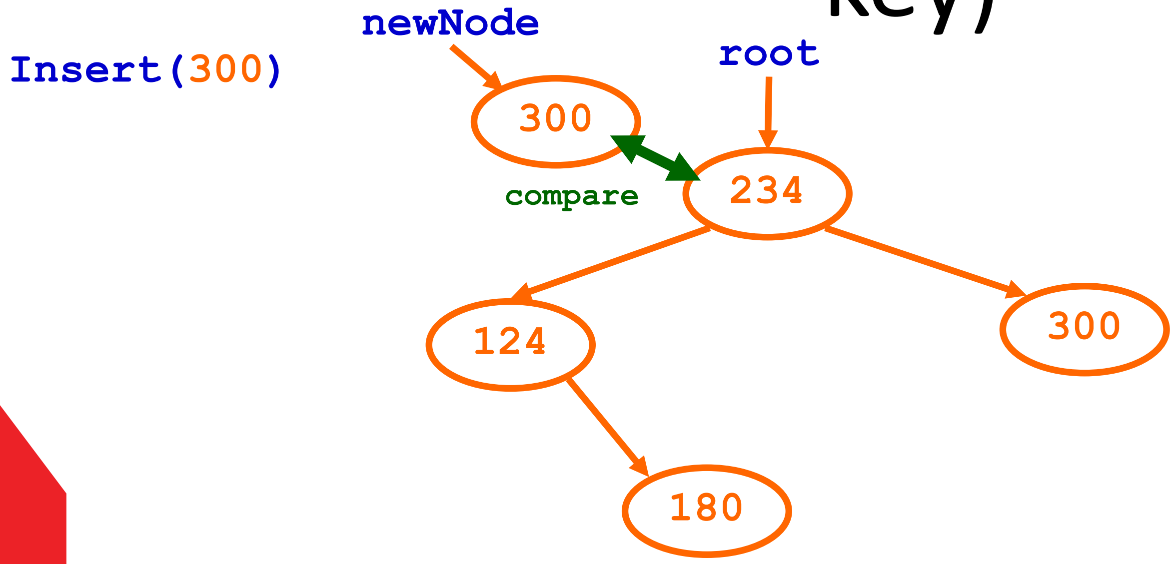
Insert(124)



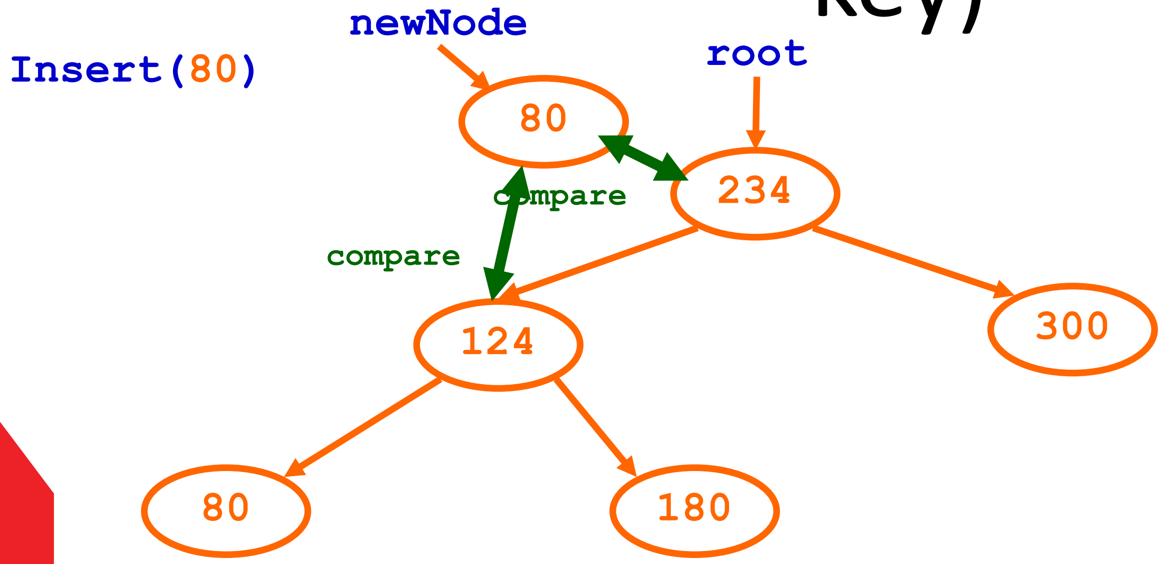
BST Insert Animation (for integer key)



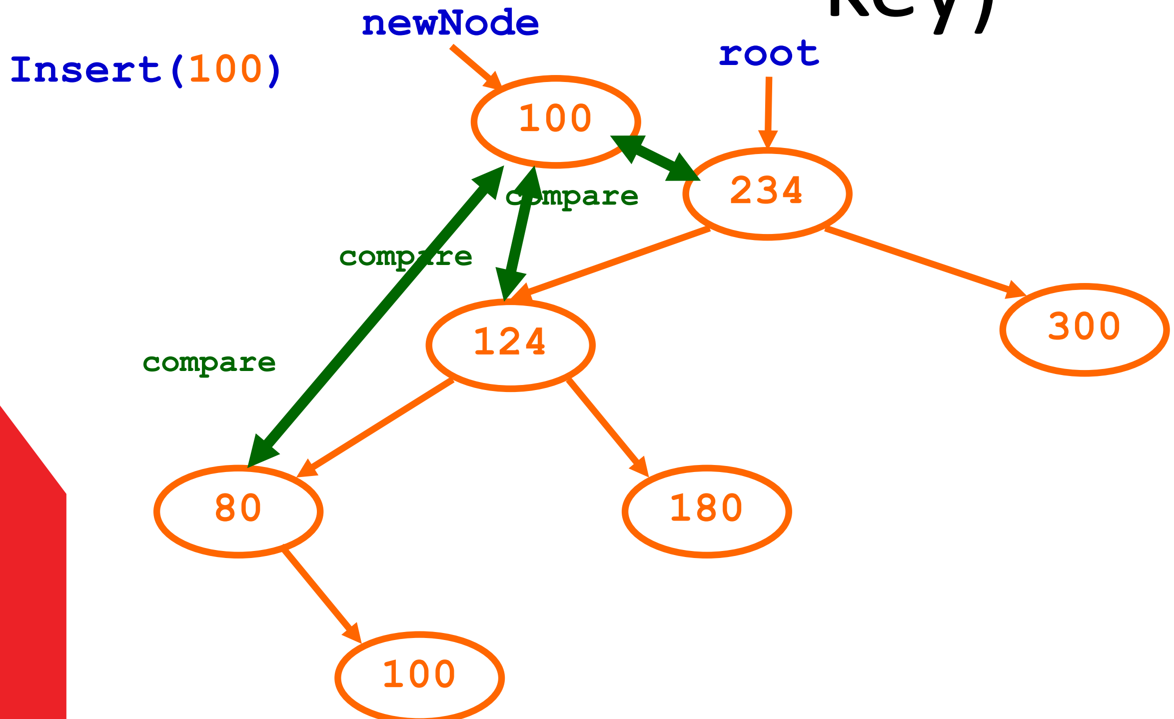
BST Insert Animation (for integer key)



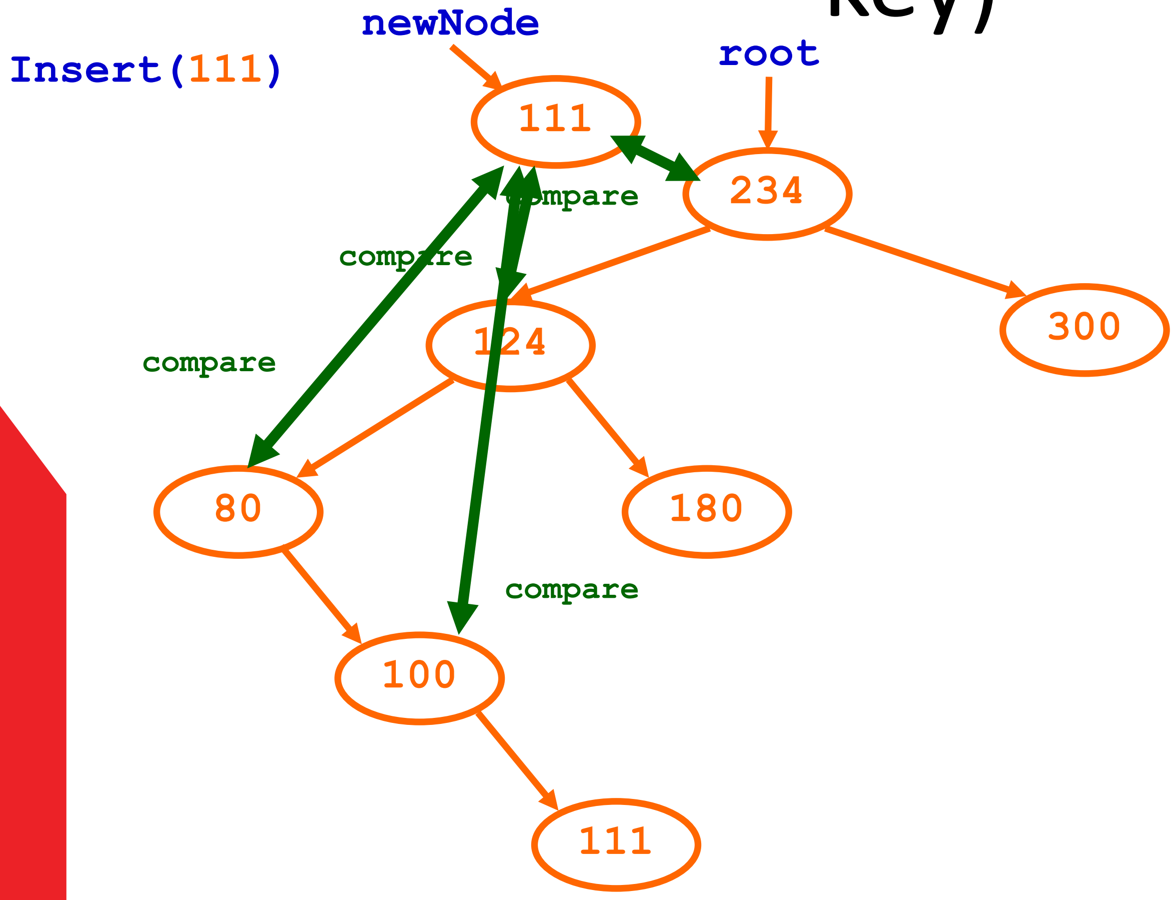
BST Insert Animation (for integer key)



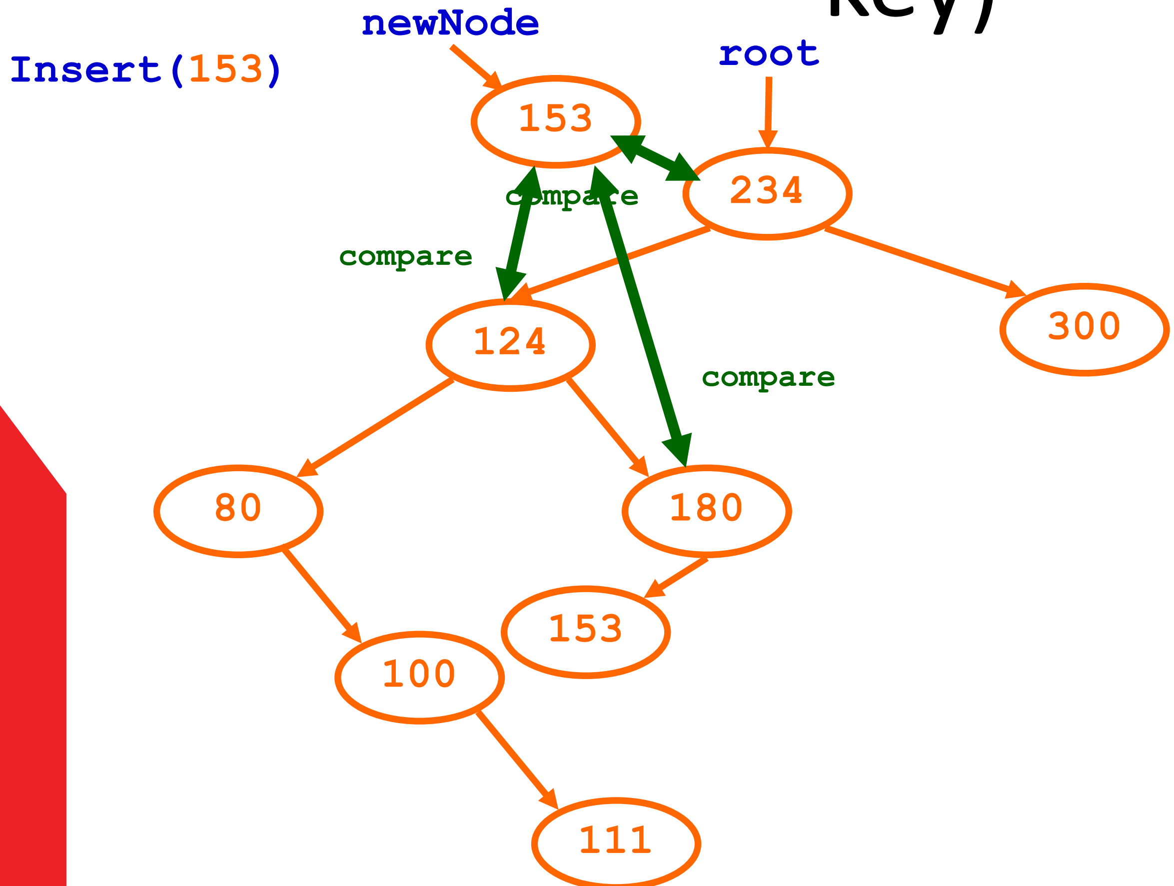
BST Insert Animation (for integer key)



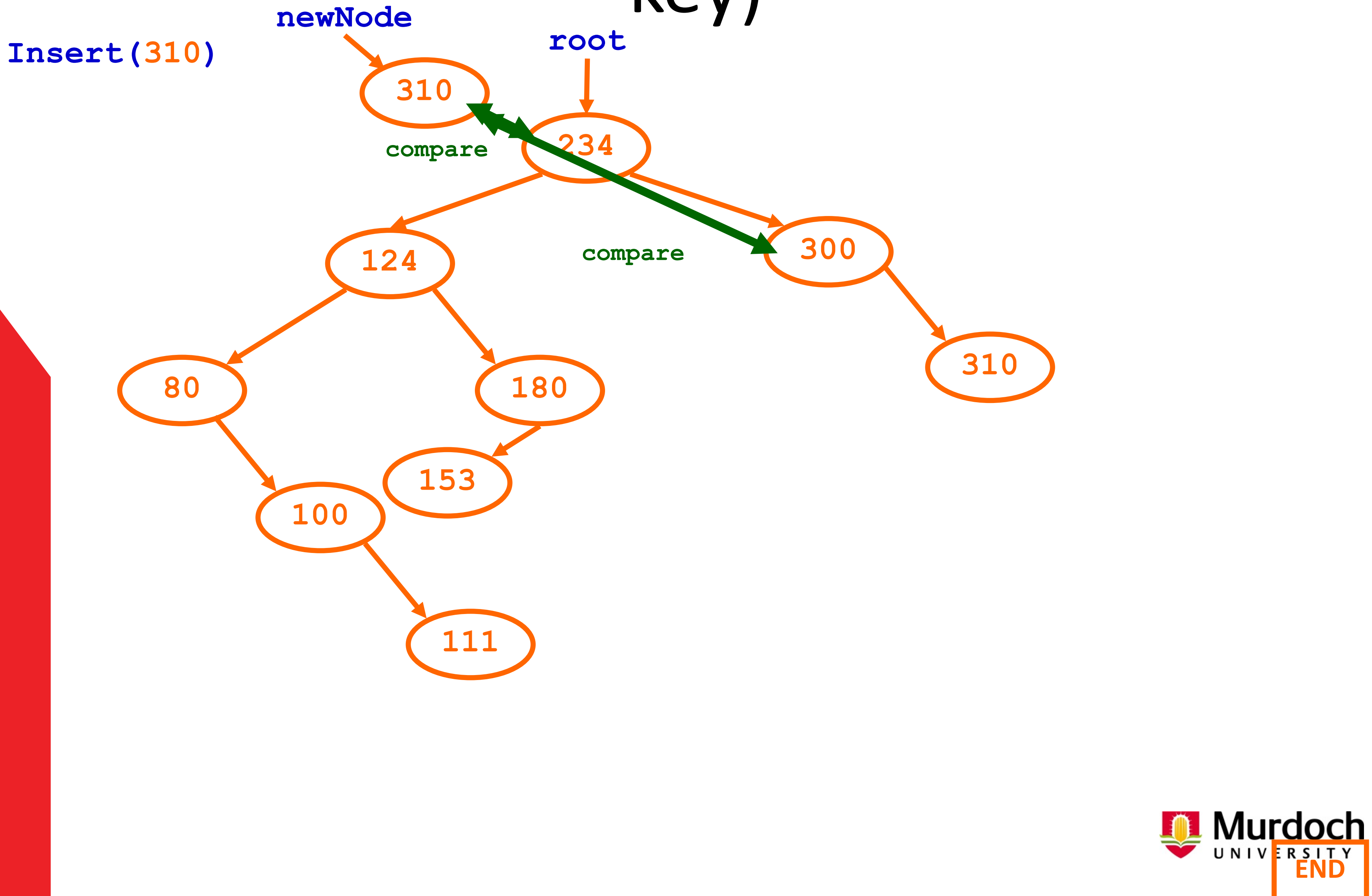
BST Insert Animation (for integer key)



BST Insert Animation (for integer key)



BST Insert Animation (for integer key)



BST Insert

- Insert (newdata) [1]
- Get memory for a newNode
- Place new data in the newNode
- IF root is NULL
- root = newNode [2]
- ELSE
- Insert (newNode, root) [3]
- ENDIF
- END Insert

Murdoch UNIVERSITY

```
•   Insert (newNode, parent) [1]
•       IF newNode’s data < parent.data
•           IF parent has no left child
•               parent.leftLink = newNode
•           ELSE
•               Insert (newNode, parent.leftLink)
•           ENDIF
•       ELSE // what happens if newNode data == parent.data?
•           IF parent has no right child
•               parent.rightLink = newNode
•           ELSE
•               Insert (newNode, parent.rightLink)
•           ENDIF
•       ENDIF
•   END Insert
```

Murdoch UNIVERSITY

BST Problem

- The trouble with the ordinary BST, is that if ordered data is inserted, you end up with a linked list.
- This means that searching a BST is O(log n) on average at best, but has a worst case complexity of O(n). (O(h))
- This problem is solved by using a *balanced* BST instead of the simple BST.
- However, the solution comes at the cost of more difficult algorithms.
- Which in turn means that programming, testing, debugging and maintaining becomes more time consuming.

Murdoch UNIVERSITY

AVL Trees

- Invented by **A**delson-**V**elski and **L**andis (so AVL) [1]
- It is a height balanced tree. [2] – see separate diagram.
- In other words the height of the left and right subtrees is never allowed to differ by more than 1.
- This ensures that the complexity of a search remains at O(log n).
- The height of a subtree is defined recursively as:

```
IF the tree is empty
   height = -1
ELSE
   height = 1 + max(height(leftLink), height(rightLink)
ENDIF
```

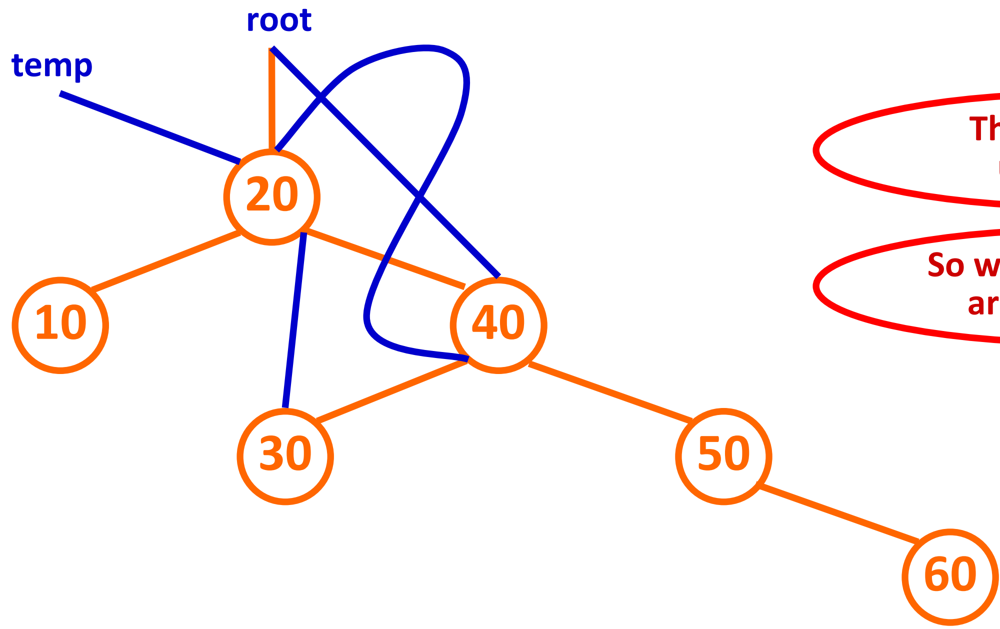
Insertion into an AVL Tree

- Insertion is done the same as for an ordinary BST.
- But if the height is unbalanced, the insertion is followed with a rebalance:

```
Insert (newNode, parent)
    IF newNode's data < parent.data
        IF parent has no left child
            parent.leftLink = newNode
        ELSE
            Insert (newNode, parent.leftLink)
            RebalanceBelowLeftOf (parent)
        ENDIF
    ELSE
        IF parent has no right child
            parent.rightLink = newNode
        ELSE
            Insert (newNode, parent.rightLink)
            RebalanceBelowRightOf (parent)
        ENDIF
    ENDIF
END Insert
```

Murdoch UNIVERSITY

AVL Tree Insert Animation

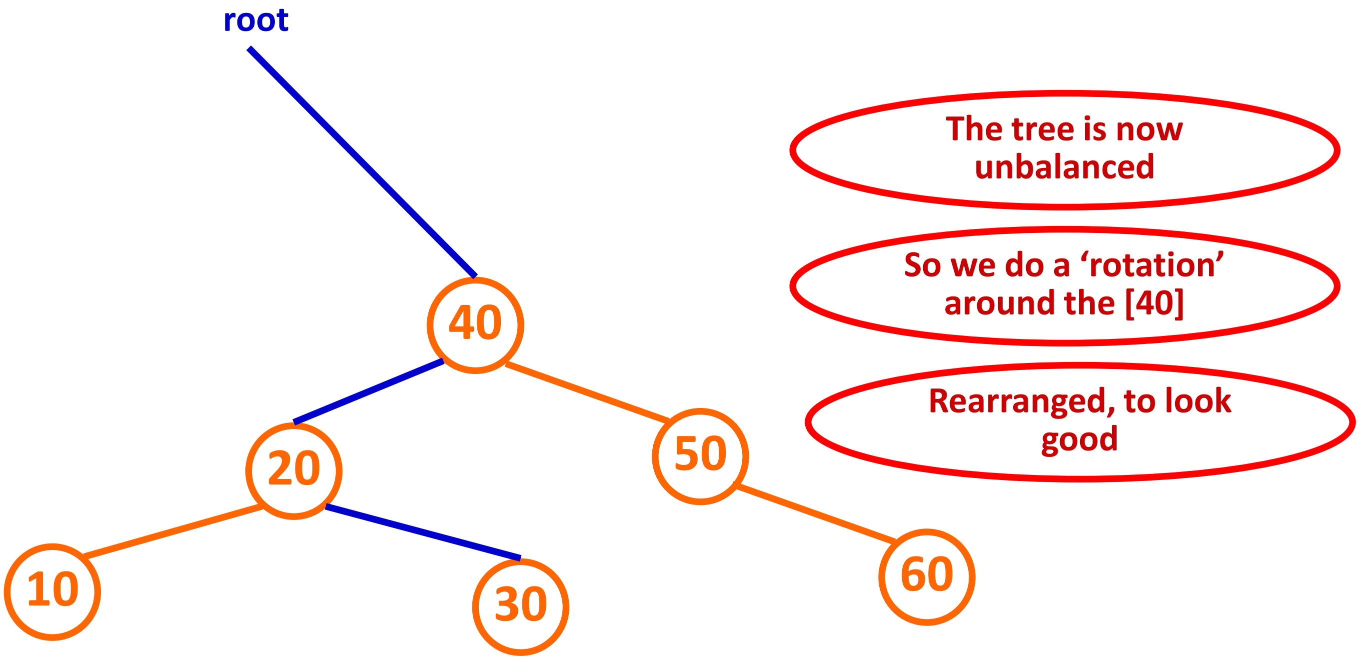


The tree is now unbalanced

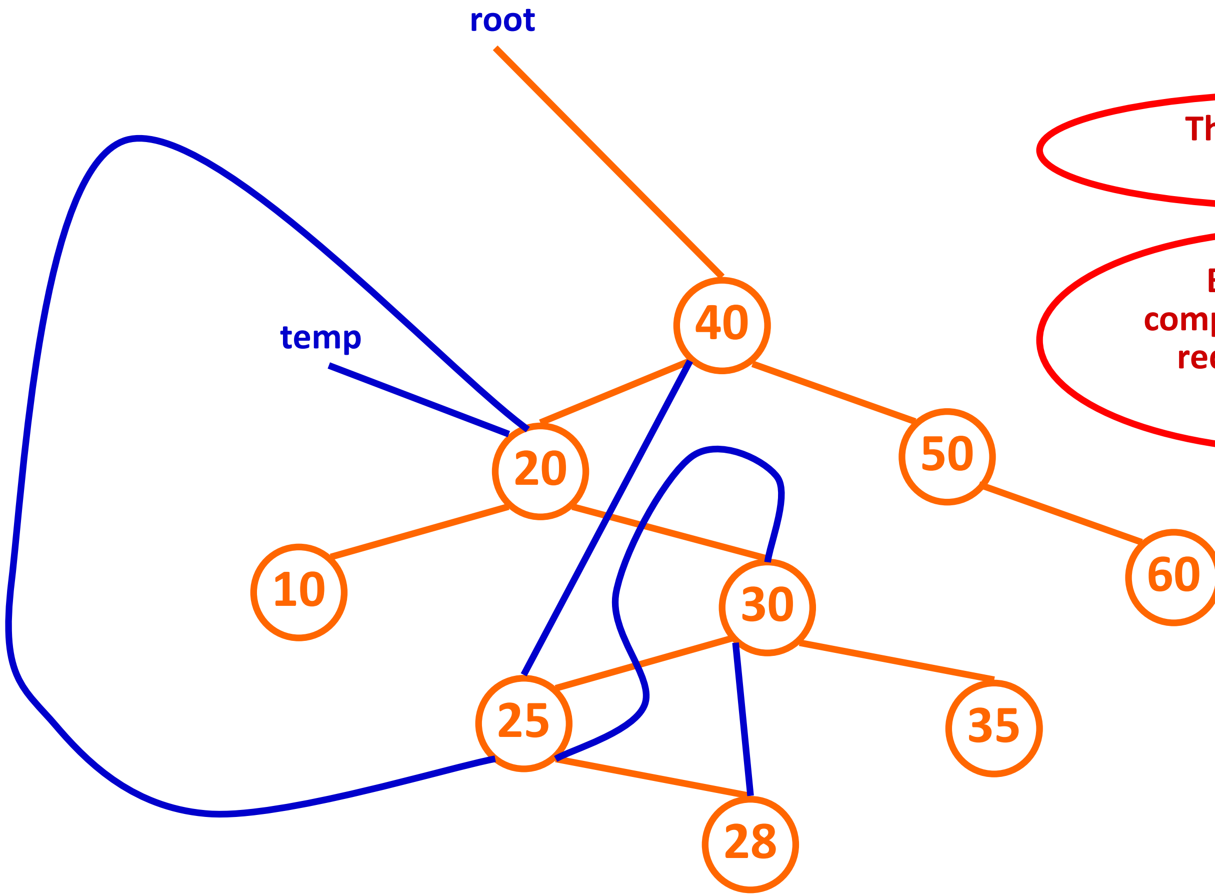
So we do a 'rotation' around the [40]

END

AVL Tree Insert Animation



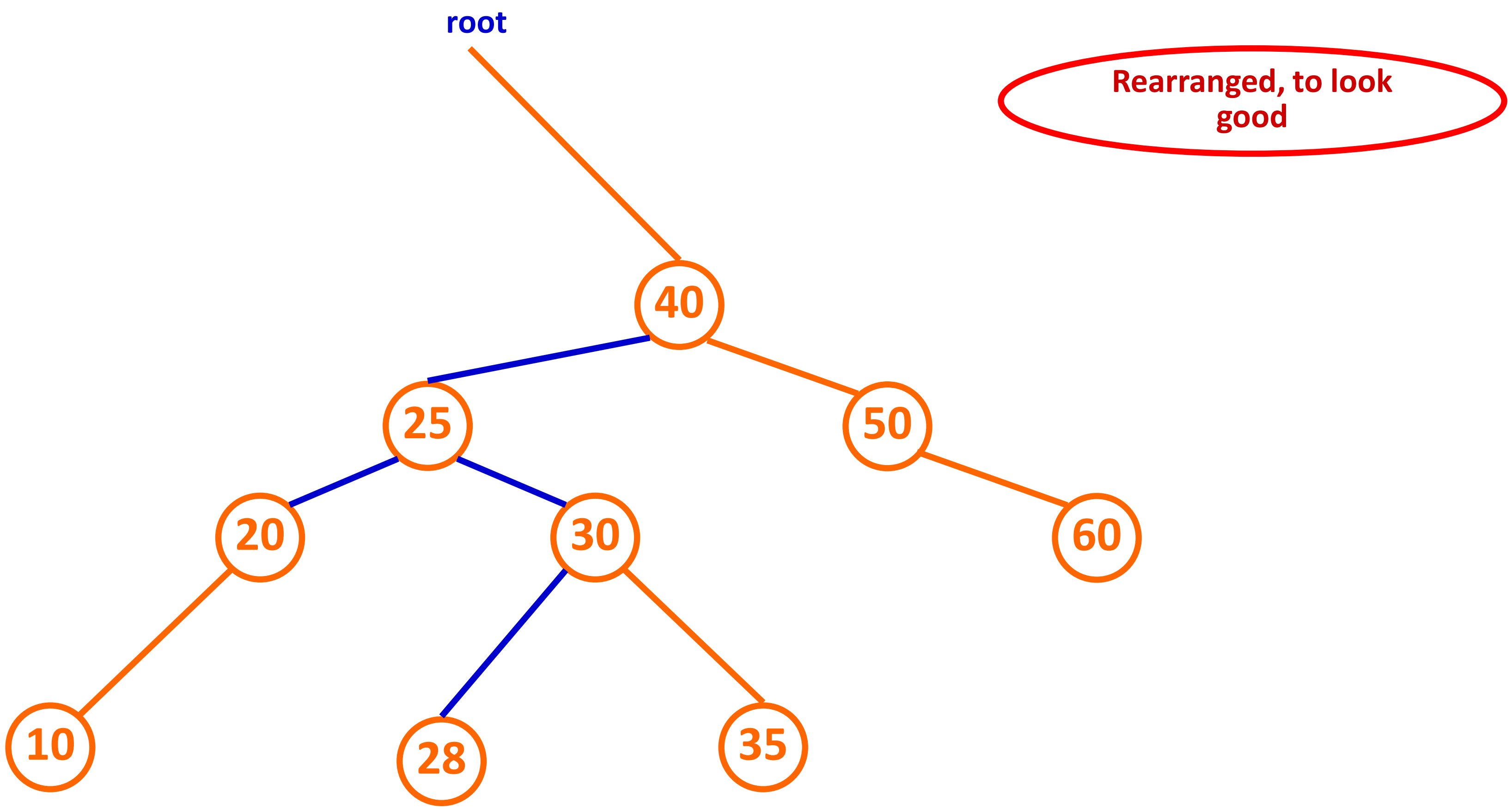
AVL Tree Insert Animation



The tree is again unbalanced

But this more complicated situation requires a *double* rotation

AVL Tree Insert Animation



Rebalancing and Rotations

- Inserting into an AVL tree can result in the tree becoming unbalanced.
- The part of the tree that is unbalanced is going to be somewhere on the tree from the insertion point to the root of the tree as only these subtrees are affected by the insertion.
- Rebalancing needs to be carried out to maintain the AVL property.
- The rebalancing is done by rotation operations.
- For example a single rotation swaps the role of the parent and child maintaining the search order. For a number of cases, single rotation doesn't work so double rotations are used. You are encouraged to find out how these operations work on your own. It is not examinable this semester. [1]
- What *is* examinable is the ability to draw the tree *after* the rebalancing, so that is what you need to be able to do.
- You are also encouraged to find out more about Red-Black trees. These are good alternatives to AVL trees.

The Programs [1]

- **`BTreeSolver`** shows the resulting BST, AVL Tree, Max Heap and/or Min Heap after insertions (which can be randomly generated or chosen).
- **`HeapSort`** shows the steps involved in a heap sort.
- **`MTreeSolver`** shows the resulting Multiway tree, BTree and/or B Plus Tree after insertions. These are covered in the next lecture.
- **`Graphs`** allows you to build graphs and then view information about the graphs (future lectures).

Murdoch UNIVERSITY

Readings

- Textbook: Chapter on Binary Trees, particularly the section on Binary Search Trees.
  - Should go through the programming example at the end of the chapter.

Textbook: Chapter on Recursion

Further exploration

- In the lab/assignment, you would normally be asked to provide a rational for your data structures. In this video link below (from an MIT unit on Introduction to Algorithms) for BST justification one particular example is used.
- MIT Lecture:
  - https://www.youtube.com/watch?v=9Jry5-82I68&index=5&list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb. In the video, the tree algorithm is modified to cater for a new requirement. This approach shouldn't be used– see Open Closed Principle. Think of a better solution. Other than that, the video explains the BST and its use very well.

Murdoch UNIVERSITY

Further exploration

- Reference book, Introduction to Algorithms. For further study, there are many tree and tree algorithms described in the reference book. For this unit, the lecture notes, practical work and the textbook is sufficient.
- Optional – recurrence trees https://www.youtube.com/watch?v=8F2OvQIlGiU
- An earlier textbook used in this unit (some years ago) is a better reference to some of the more interesting Tree (and graph) data structures like AVL trees, Red Black trees and AA trees. The book is available in the library. It is "Algorithms, Data Structures, and Problem solving using C++" by Mark Weiss.

- AVL trees .. from an MIT unit Introduction to Algorithms
- MIT Lecture:
  - https://www.youtube.com/watch?v=FNeL18KsWPc&index=6&list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb
  - MIT Tutorial (different to a lab, no computers) https://www.youtube.com/watch?v=IWzYoXKaRIc&list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb&index=29
  - https://www.youtube.com/watch?v=r5pXu1PAUkI&list=PLUl4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb&index=28

Murdoch UNIVERSITY

Murdoch UNIVERSITY

Data Structures and Abstractions

Multiway Trees

Lecture 30

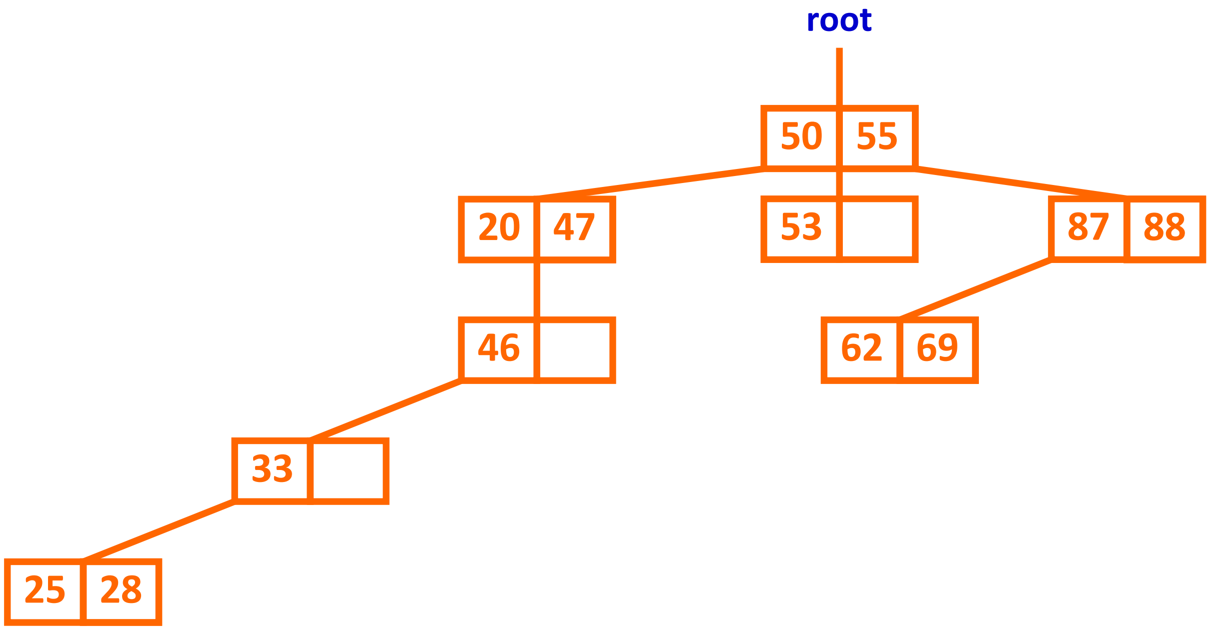
Multiway Trees

- Multiway trees are trees that store more than one piece of data in a node and more than two links.
  - Note: a Binary tree stores one piece of data.
- A 3-way tree stores up to 2 items of data per node.
  A 4-way tree stores up to 3 items of data per node,
  etc.
- Insertion is done in the same way as with a simple BST.
- Of course this means that, like a simple BST, it is possible to end up with a linked list.
- Reminder: in the animations we just show storage of a single integer, but in reality trees are used to store larger amounts of information using a key. [1]

Murdoch UNIVERSITY

3-way Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)
Insert(69)
Insert(28)



END

B Trees

- B Trees are balanced multi-way trees in which a node can have up to k subtrees.
- Suitable for data storage on disks when collections are too large for internal memory.
- As for most data stores, the elements are usually records, which have a key and a value.
- The key is used to locate the node where the record is to be stored.

B Tree Definition

- Formally, a B Tree of order m is a multi-way tree in which:
  - the root is either a leaf or has at least two subtrees;
  - each leaf node holds at least m/2 keys;
  - each non-leaf node holds k-1 keys and k pointers to subtrees where m/2 <= k <= m;
  - all leaves are on the same level.
- m is normally large (50-500) so that all the information stored in one block on disk can fit into one node.

Murdoch UNIVERSITY

Insertion into a B Tree

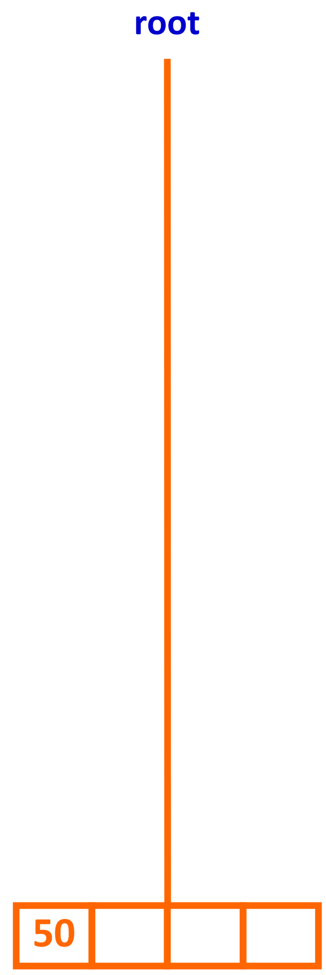
- Insertion of a key (and its record) is always done at a leaf node.
- This may cause changes higher up the tree.
- The method is:
  1. Locate: Do a search to locate the leaf in which the new record should be inserted.
  2. Insert:
     a) If the leaf has room, insert the record, in order of key.
     b) If the node if full, 'split' it and move the record with the median key upwards.
     c) Repeat (b) until either a non-full node is found, or root is reached.
     d) If the root is full, split it and create a new root node containing one key.

Murdoch UNIVERSITY

Insert(50)

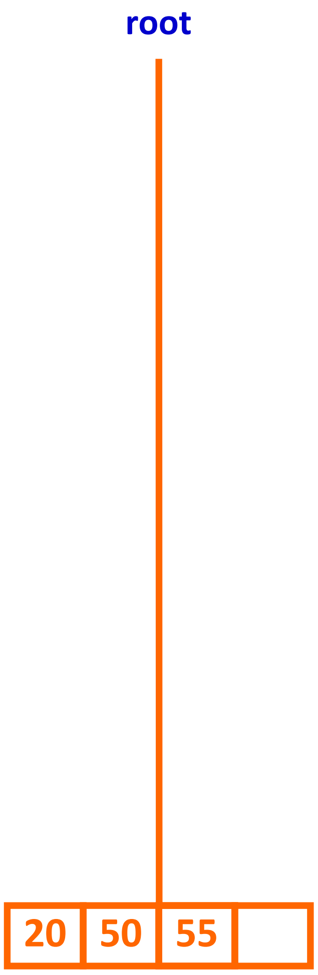
Insert(20)

5-way B Tree Insert Animation



5-way B Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)



5-way B Tree Insert Animation

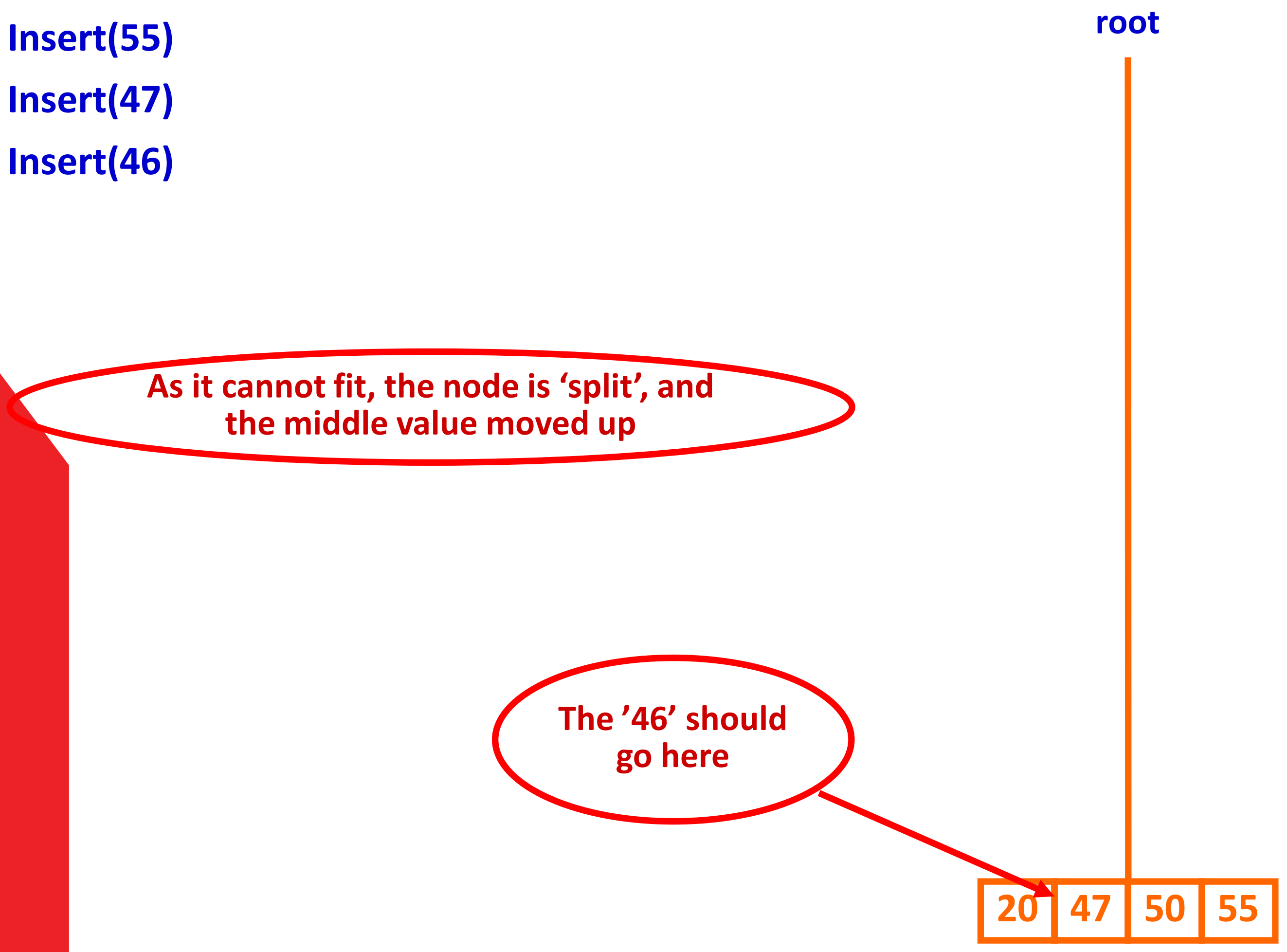
Insert(50)

Insert(20)

Insert(55)

Insert(47)

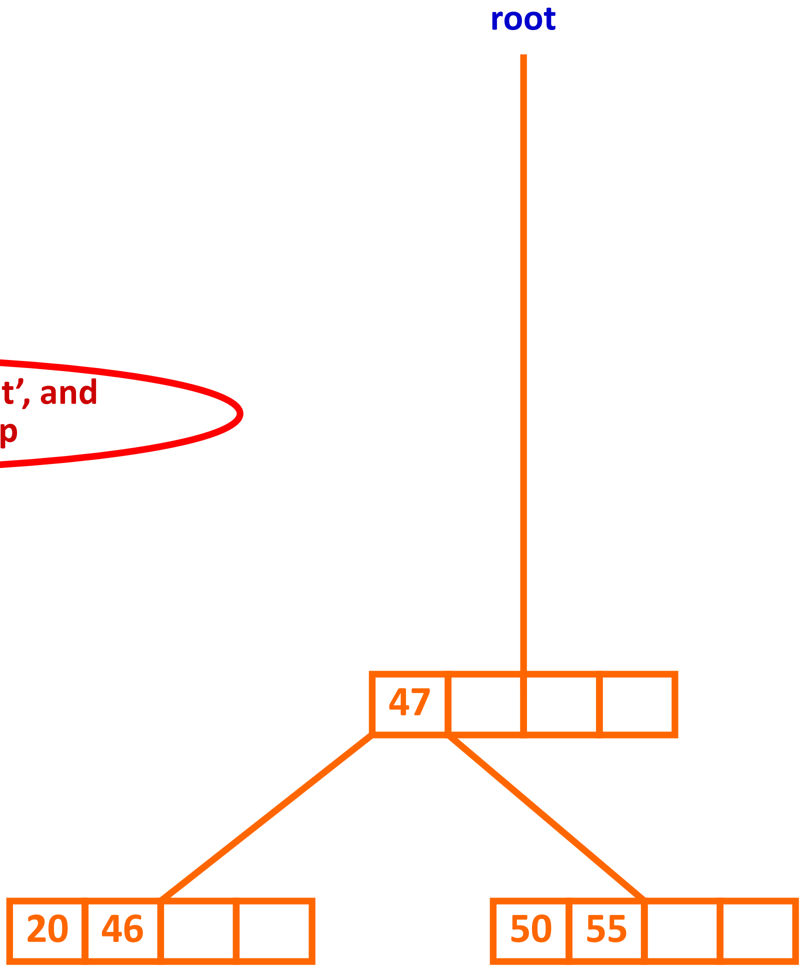
Insert(46)



Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)

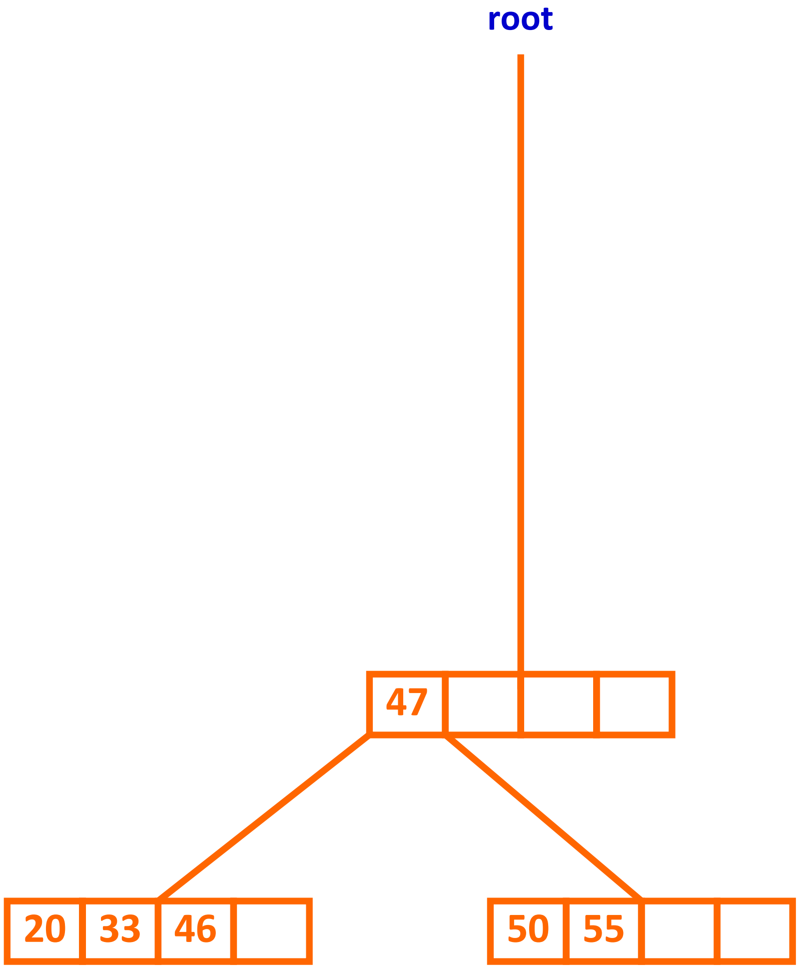
5-way B Tree Insert Animation

As it cannot fit, the node is 'split', and
the middle value moved up



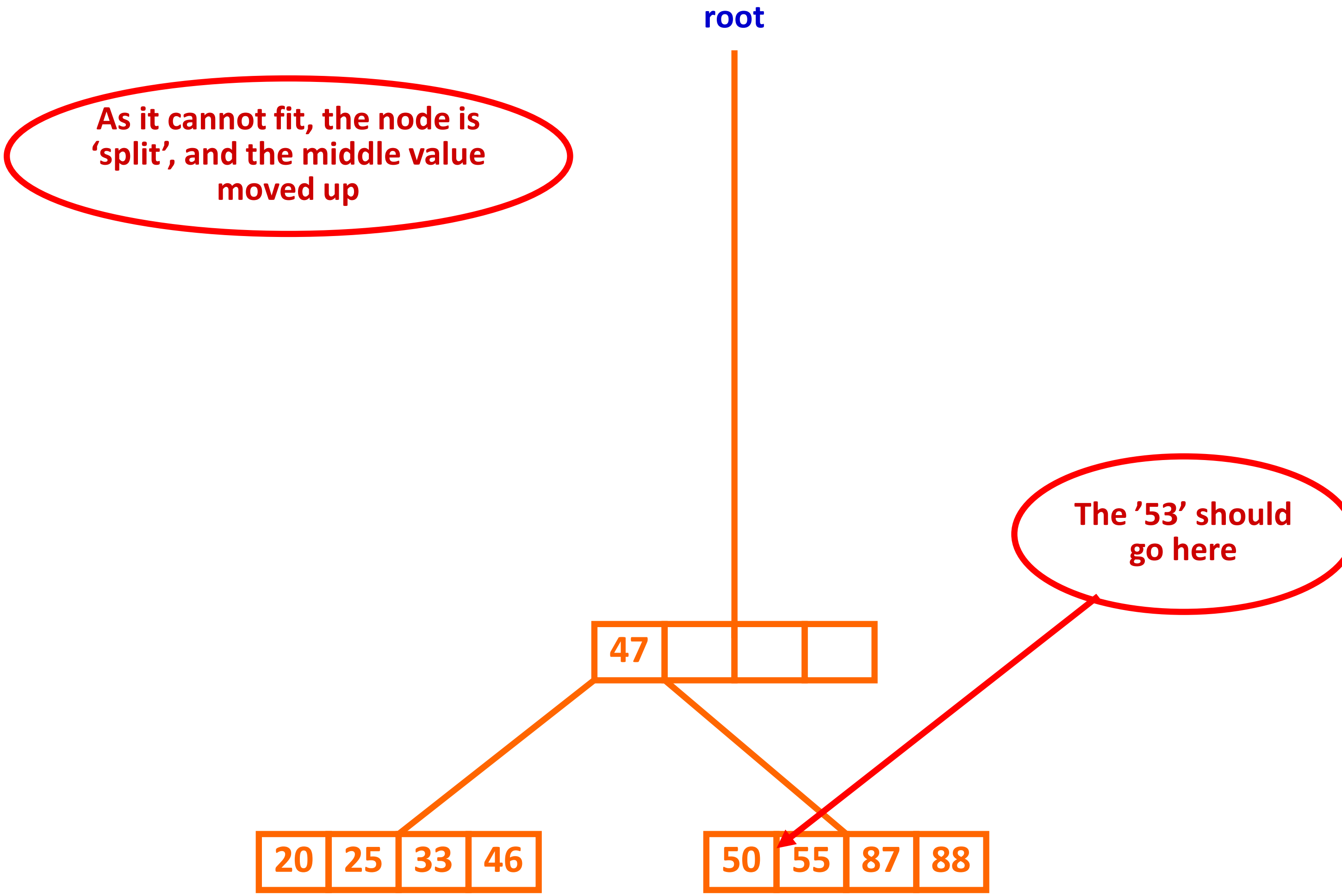
5-way B Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)



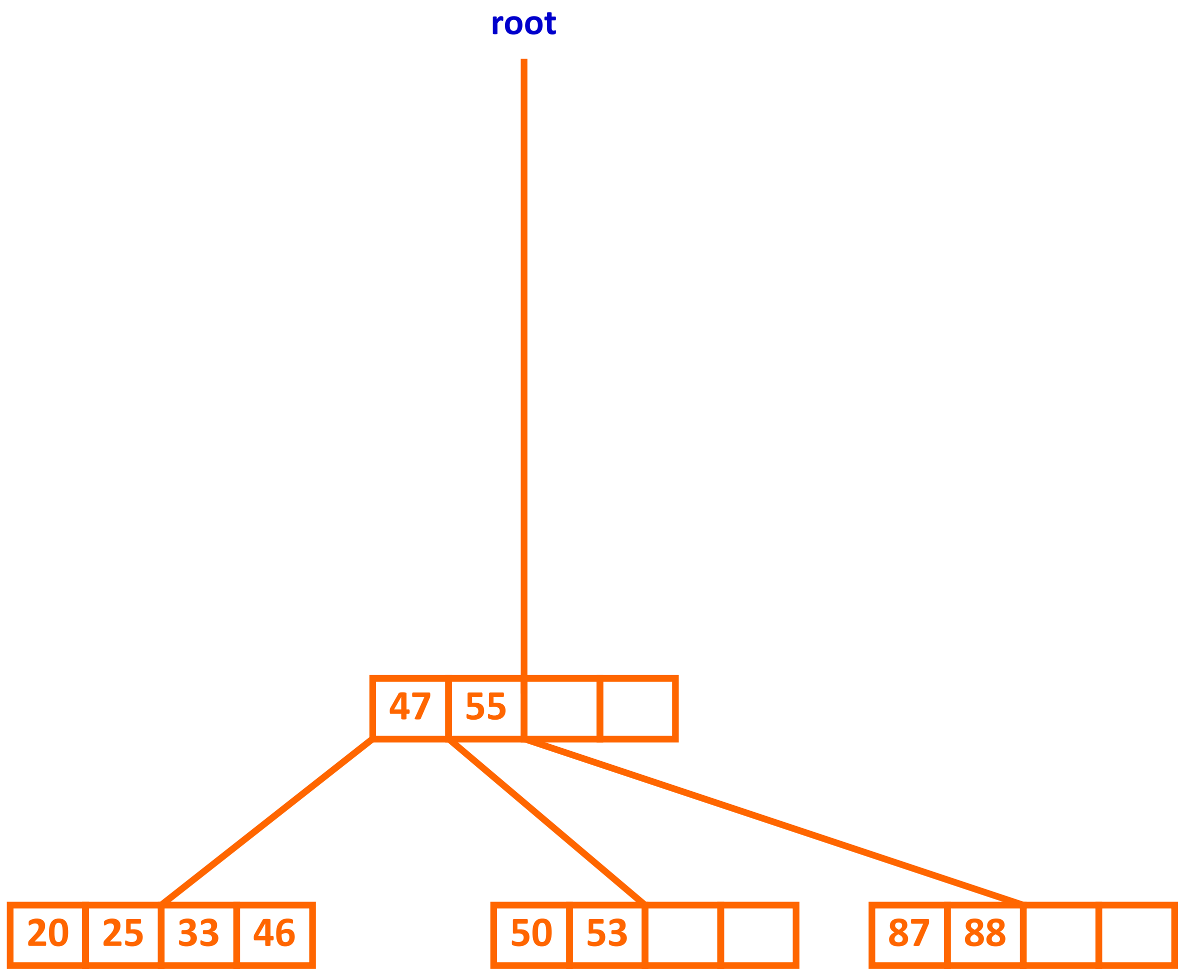
5-way B Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)



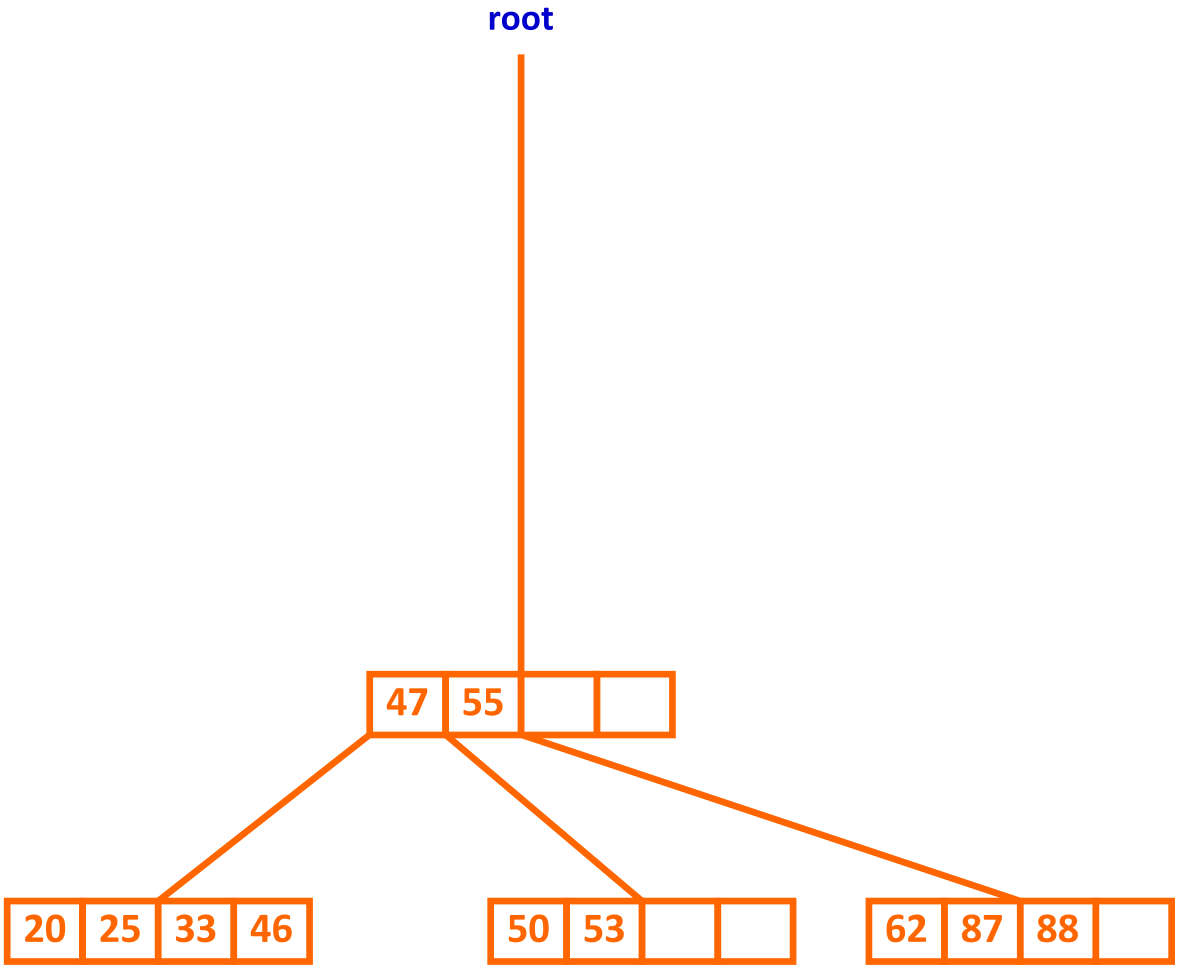
5-way B Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)



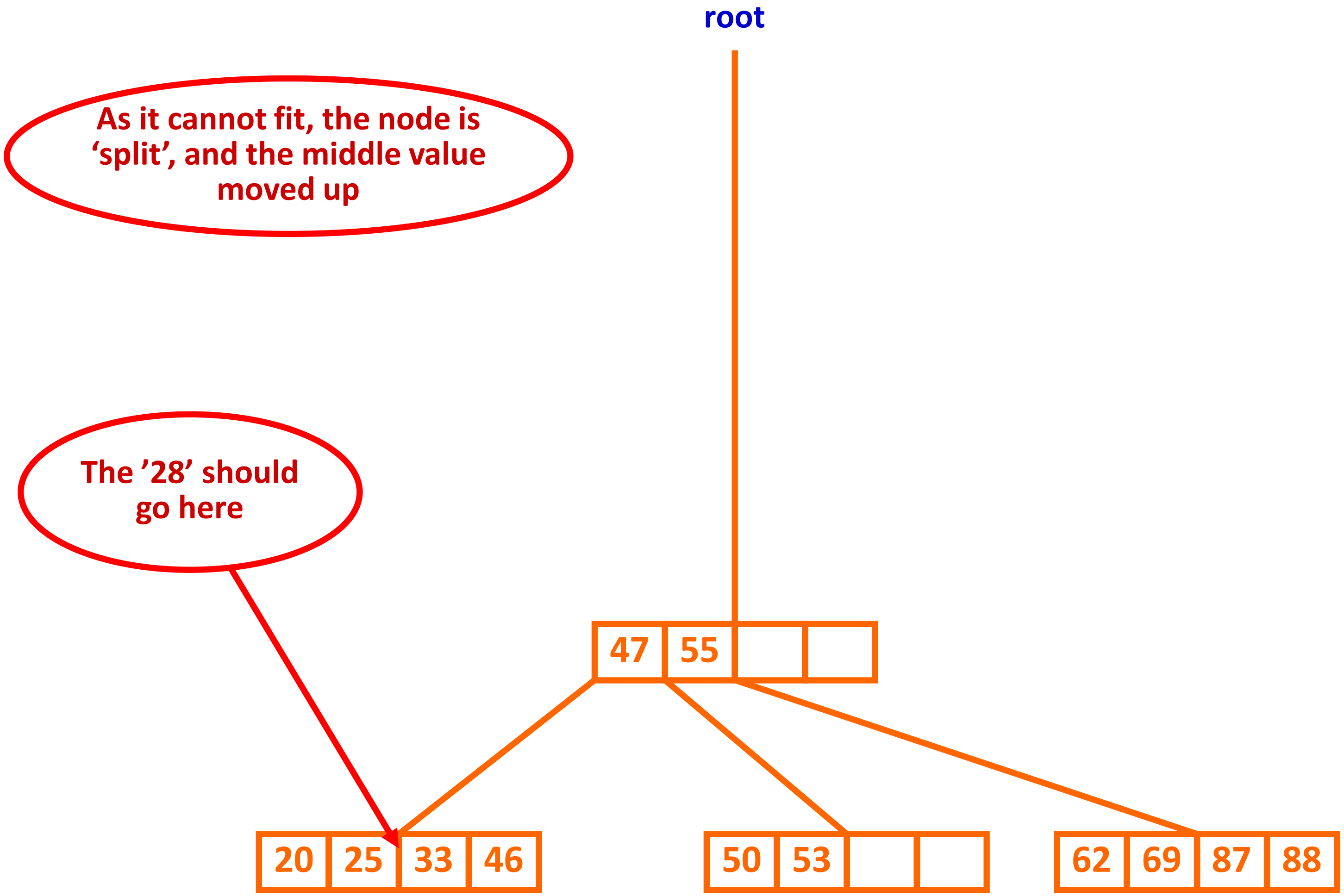
5-way B Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)
Insert(69)



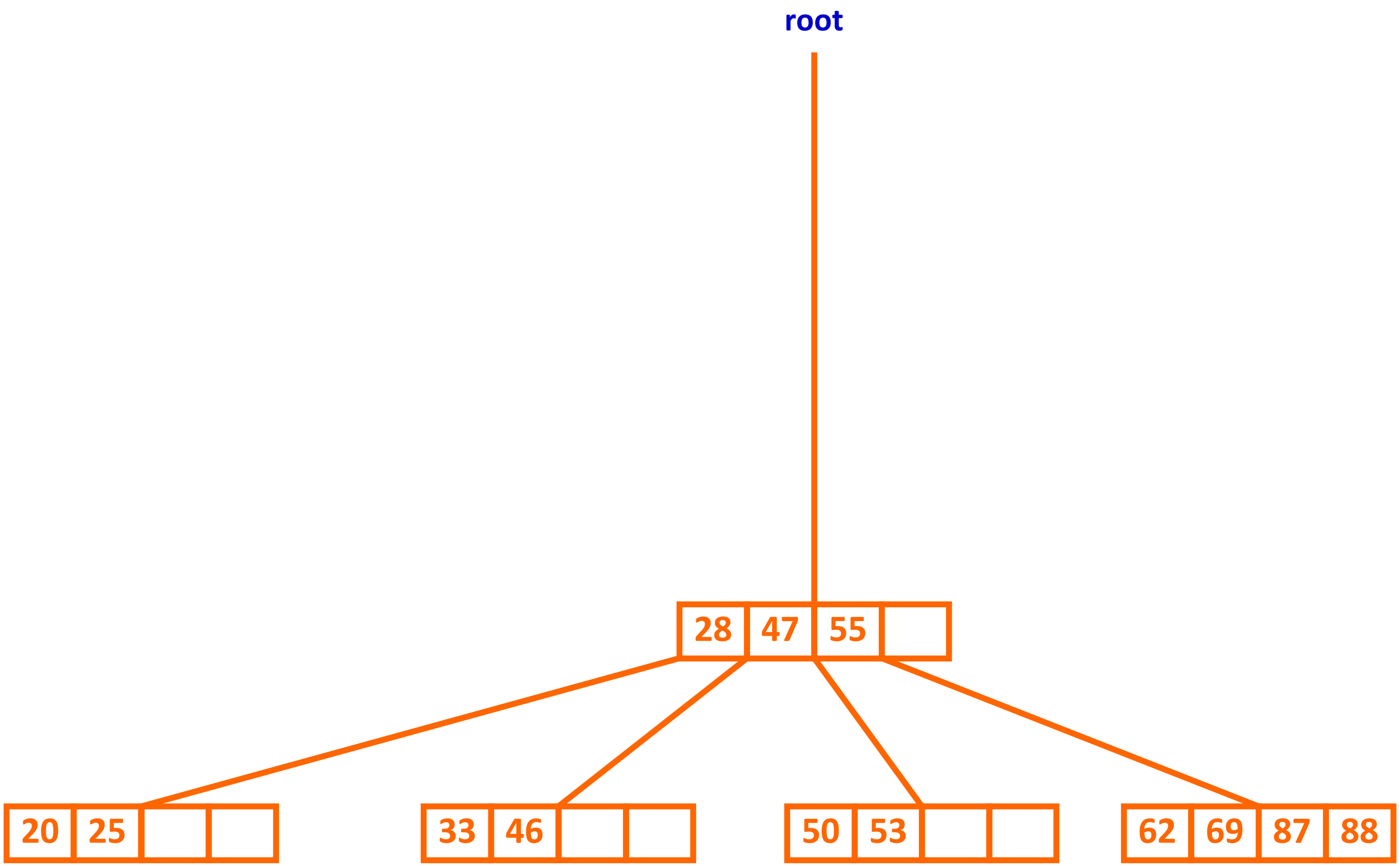
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)
Insert(69)
Insert(28)

5-way B Tree Insert Animation



5-way B Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)
Insert(69)
Insert(28)



END

Multiway Tree vs B Tree

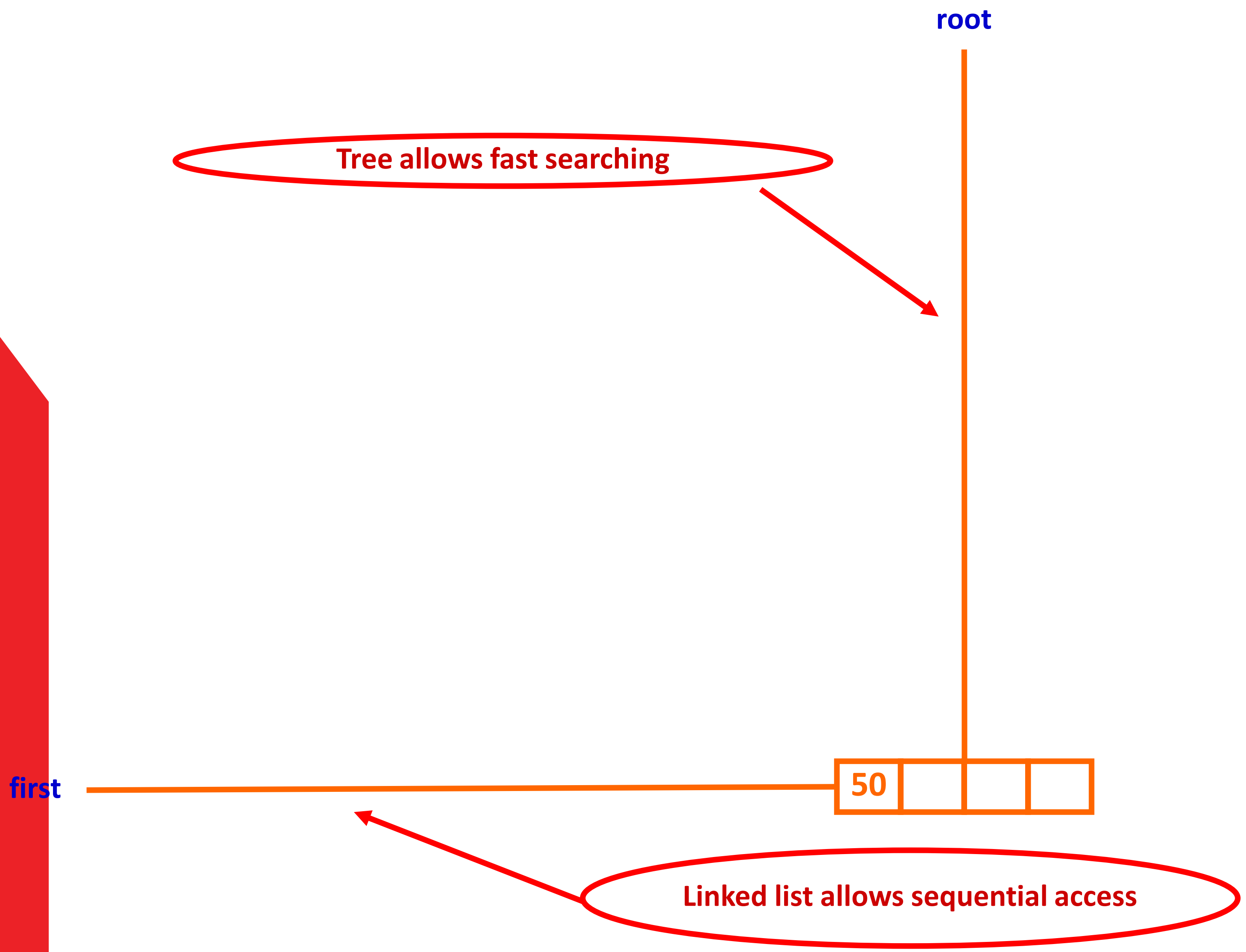
- The B Tree is clearly more efficient in terms of space.
- The B Tree is balanced and therefore has a lower search time.
- The B Tree, however, is more complicated to code.
- If search time is important (as with a database or list of objects in the scene of a game) the use of B Trees is essential.

B+ Trees

- Trees are good for searching, but have poor sequential access.
- Some databases require both types of processing, for these one uses a B+ tree.
- A B+ tree is a B Tree where only the keys are stored in the tree, all the data actually resides in the leaves.
- And the leaves are all connected with a list.
- This kind of tree is particularly useful for databases that reside entirely on disk.

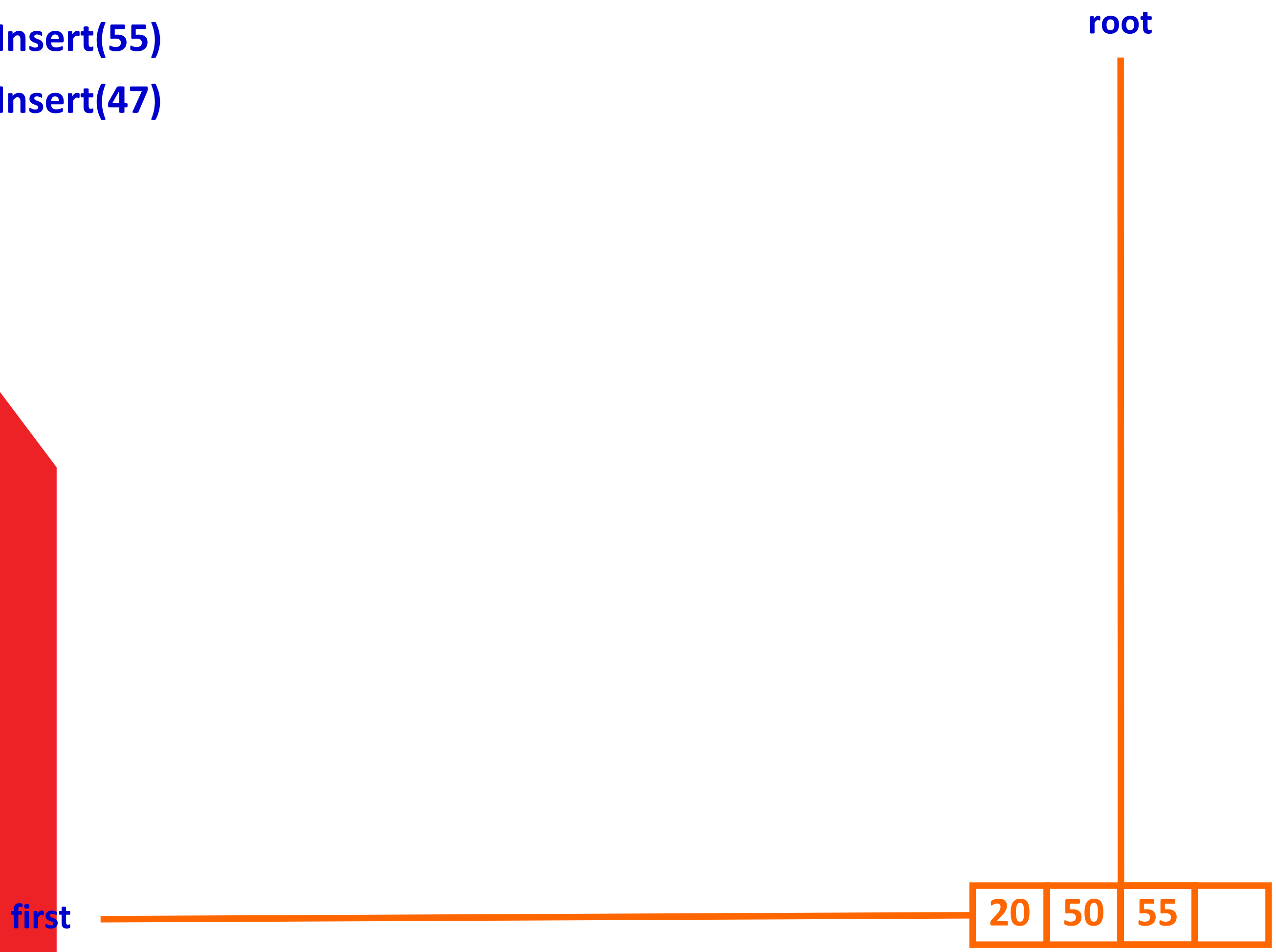
Insert(50)
Insert(20)

5-way B+ Tree Insert Animation



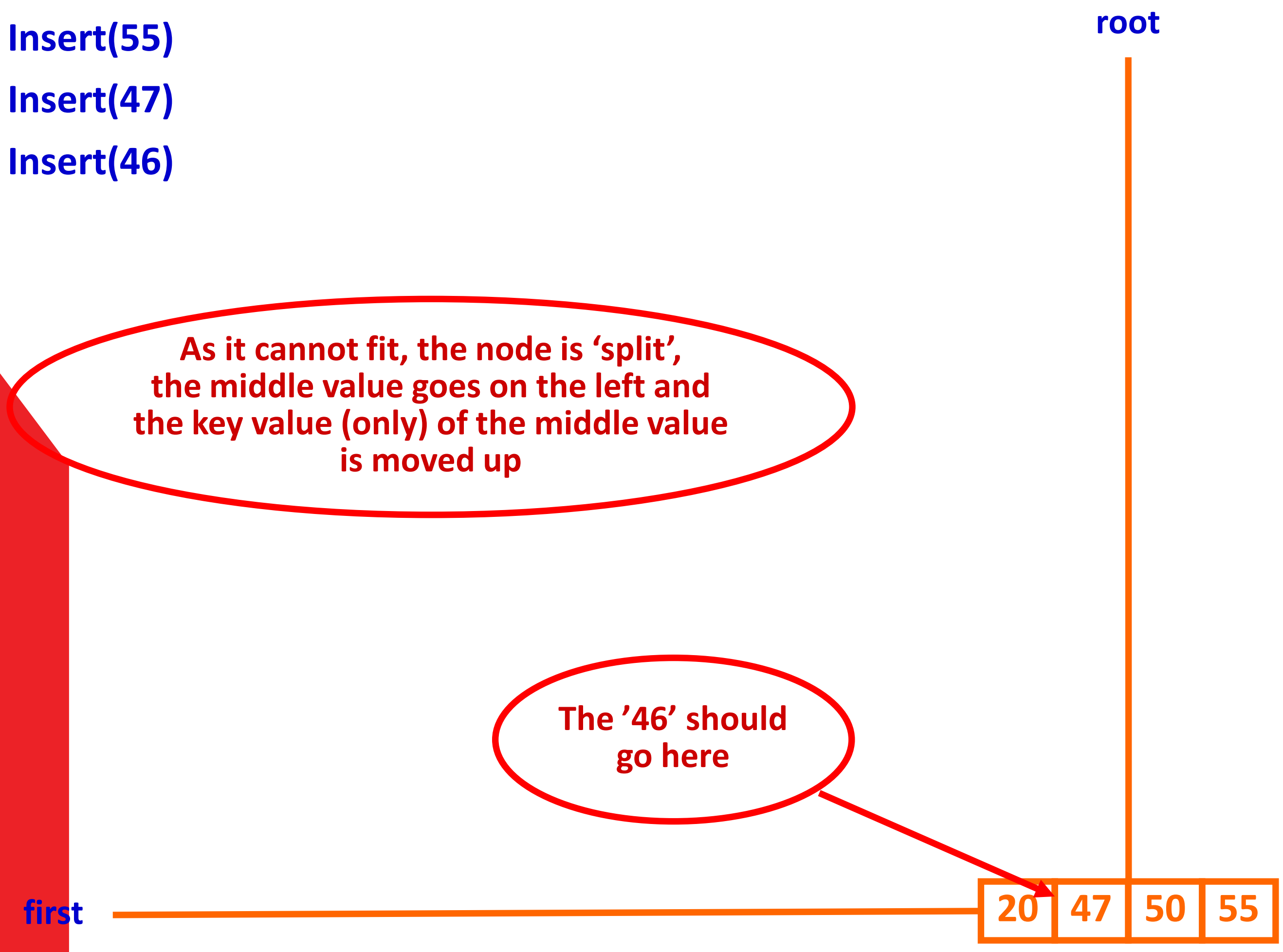
5-way B+ Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)



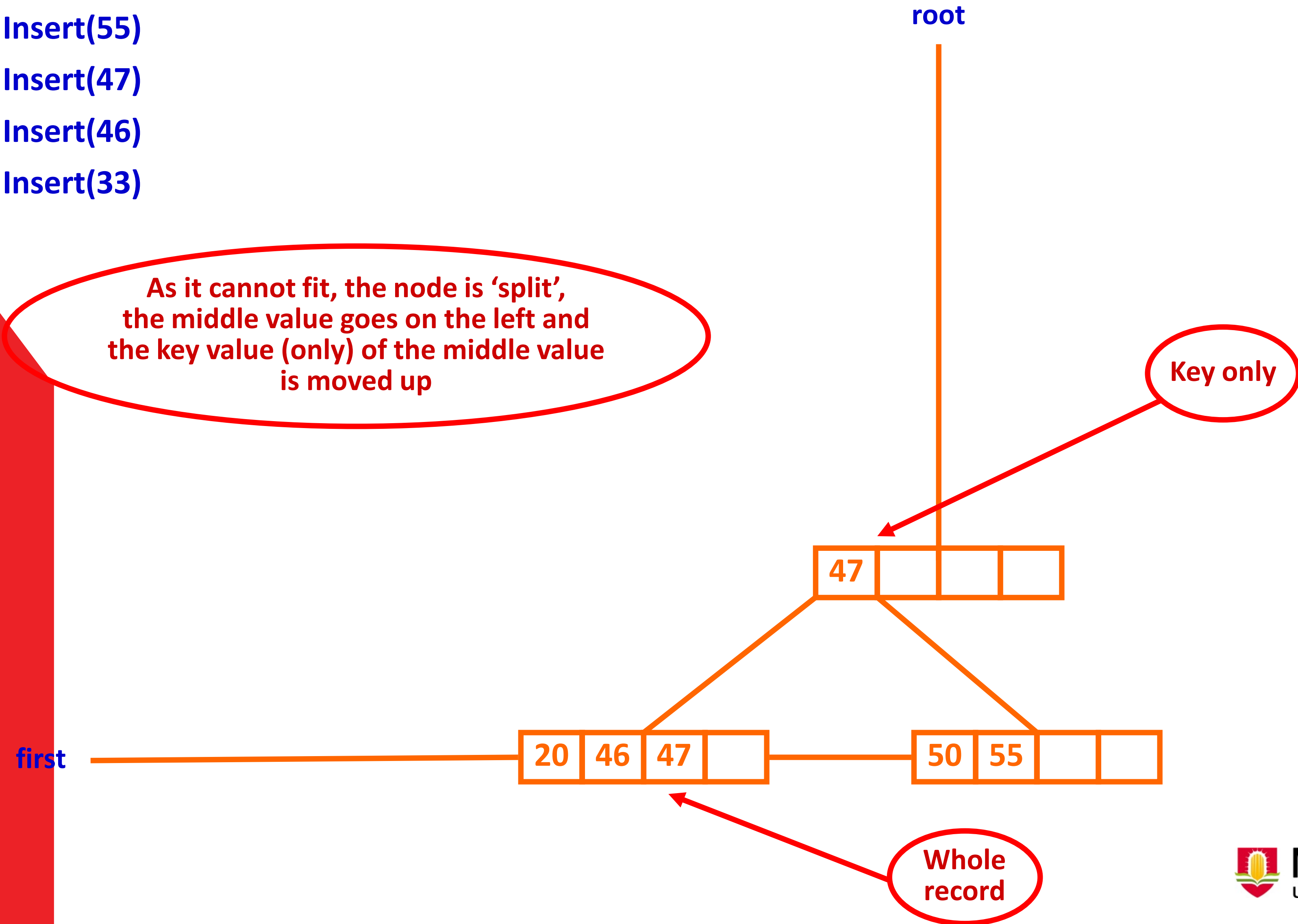
5-way B+ Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)



5-way B+ Tree Insert Animation

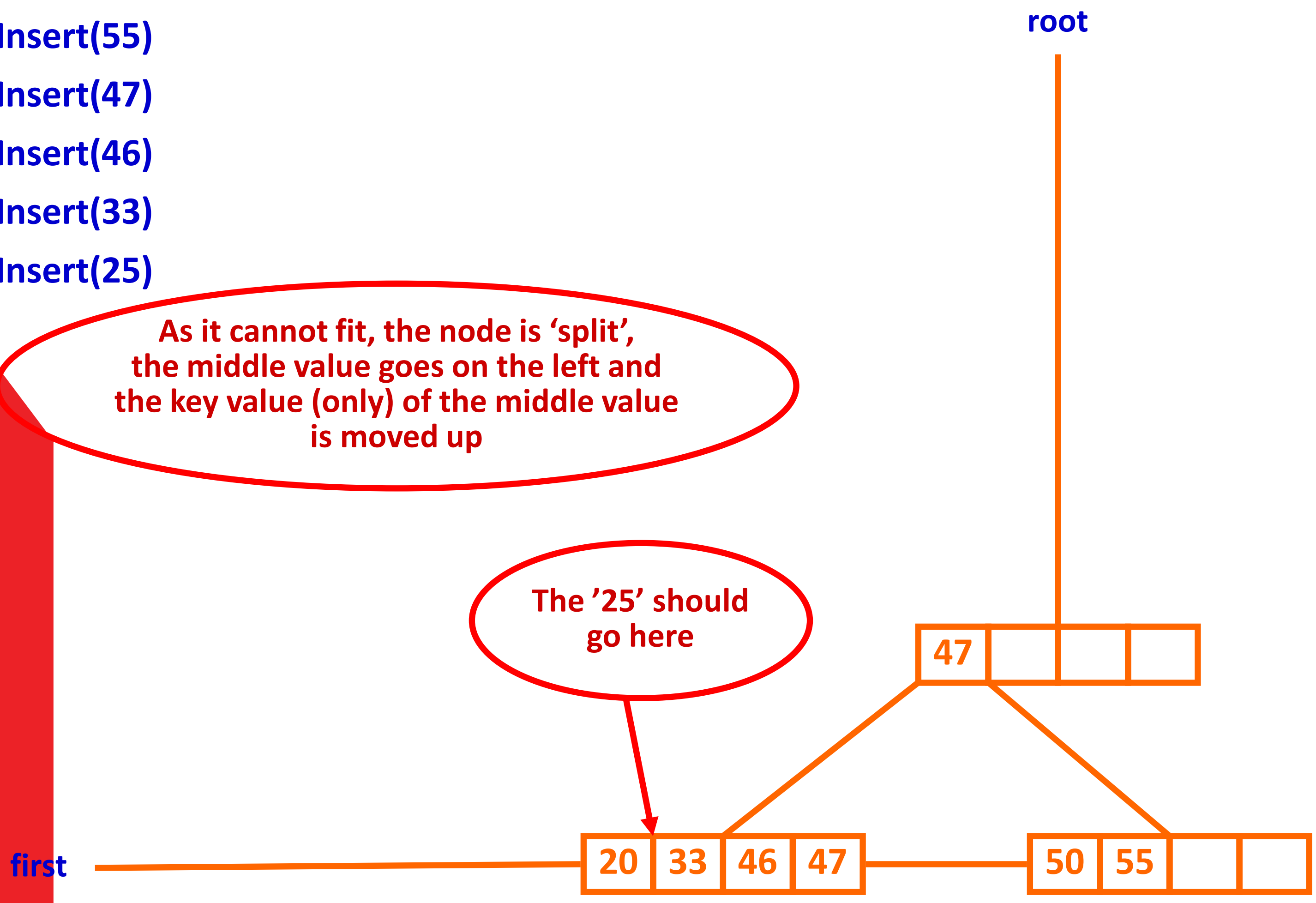
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)



5-way B+ Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)

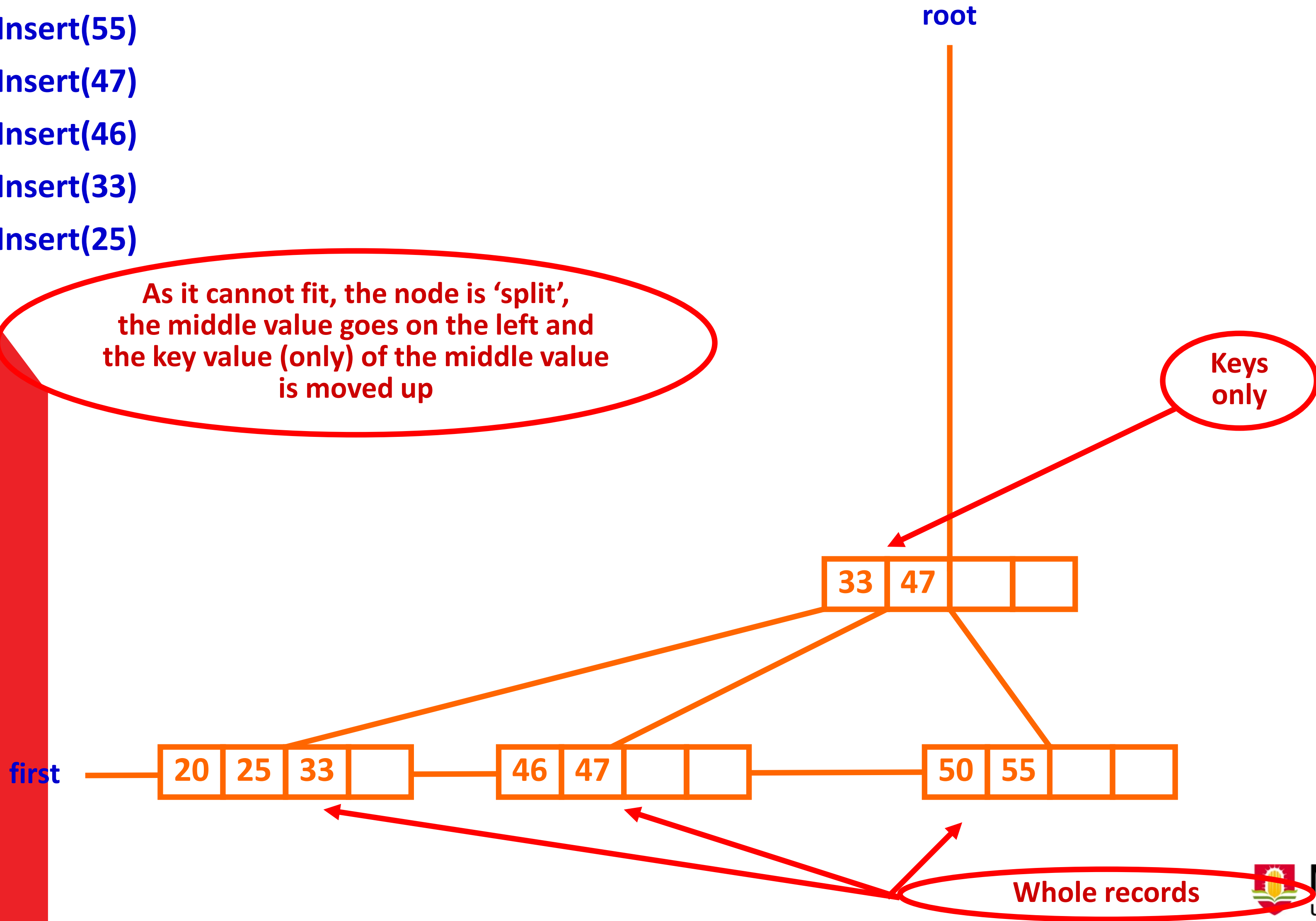
As it cannot fit, the node is 'split', the middle value goes on the left and the key value (only) of the middle value is moved up



5-way B+ Tree Insert Animation

Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)

As it cannot fit, the node is 'split', the middle value goes on the left and the key value (only) of the middle value is moved up



END

Comparison of B and B+ Trees

- They are both balanced so that operations such as Insert and Delete can be done in O(h) time where h is the height of the tree.
- B+ trees also allow for fast sequential processing.
- B+ trees store the key only in RAM, not the whole record, therefore they use less RAM.
- Both can be tuned to have node sizes that allow fast disk reads.
- As B+ trees use less RAM, they can have larger nodes which improves the speed of operations based on the height.
- Both B Trees and B+ Trees have one major disadvantage in common: since any node can be up to half empty, they waste space.

Handling the Wasted Space Problem

- A way to get around this is to really treat them as ADSs, i.e. they are conceptually B Trees but are actually stored in some other way.
- For example a vector, linked list, dynamic array etc.
- The operations (Insert, Delete, Search etc) in the interface do not change, but the internal representation and code do change.
- However, it is worth noting that although there are many different ways of solving the space problem, there will *always* be a space/time/simplicity trade off.

Murdoch UNIVERSITY

Further exploration

- Reference book, Introduction to Algorithms. For further study, there are many tree and tree algorithms described in the reference book. For this unit, the lecture material is sufficient.

- An earlier textbook used in this unit (some years ago) is a better reference to some of the more interesting Tree (and graph) data structures. The book is available in the library. "Algorithms, Data Structures, and Problem solving using C++" by Mark Weiss.