



Murdoch
UNIVERSITY

Data Structures and Abstractions

Program Control in C++

Lecture 4/revision



Complete Topic 1

- You need to have completed the programming exercises and readings from topic 1.
- Completion of the previous lab exercises is particularly important.
- To be able to do the program in lab 2, you would also need to have completed the readings for topic 1.
 - Particularly chapter 3 “Input/Output” and the chapter on “Classes and Data Abstraction” from the textbook.

Control Statements

- Like almost all third generation languages, C++ has selection, iteration and call statements.
- Selection:
 - **if**
 - **switch**
- Iteration:
 - **for**
 - **while**
 - **do-while**
- Calls:
 - **functions**
 - **procedures**

Selection – IF

- `if (condition)`
- `{` *// always use these braces [1]*
- `// Do something`
- `}`
- `else`
- `{` *// always use these braces [1]*
- `// Do something else`
- `}`

Selection – Switch

- The switch statement is used when there are choices based on the value of a single variable.
- The variable has to be an *ordinal* value: an integer or boolean.

```
• switch (variable)
• {
•     case value1:
•         // do something
•         break; // do not miss this. If not required - comment
•     case value2:
•         // do something
•         break; // do not miss this. If not required - comment
•     default:
•         // do something
• }
```

Iteration - WHILE

- While loops have the condition at the beginning of the loop.
- Therefore the loop must be *primed*, i.e. the condition must be set *before* the loop commences.
- Something inside the loop must change the condition so that it ends sometime!

```
// prime the loop

while (condition)
{
    // do something
    // loop body should have code which can alter
        // condition
}
```

Iteration – Do-While

- Do-while loops execute at least once, therefore the loop does need to be primed.
- Something inside the loop, however, must change the condition.
- They are rarely useful except for menus.

```
do
{
    // do something
} while (condition);
```

Iteration – For loop

- Very versatile: can combine traditional counting with while loop condition.
- Multiple parts in the initialise and increment sections are separated by commas. Where possible avoid these.

```
for (initialising code; condition; increment)
{
    // do something
}
```

Selection and Iteration Notes

- Using the best type of selection or iterator control makes it easier to read and debug code.
- Note that **break** statements are used in switch statements and *nowhere else!*
- **continue** and **goto** statements should be avoided [1]
- These statements make the code poorly structured and therefore harder to maintain and debug. [1]

Functions

- Functions are declared above main (called prototypes) and then defined below main.
- They return a value to the variable of a defined type.
- They can have parameters and can return values within the parameters.
- Advice that user named Function names start with a capital letter and use capitals for new words eg:
FindMaximum ()
- The list of parameters used where the function is defined is called the *formal* parameter list.
- The list of parameters used where the function is called is called the *actual* parameter list.

- // function.cpp [1]
- // You need to have completed the programming exercises and readings from // topic 1.
- // Simple program showing function use
- //
- // Version
- // 001 Nicola Ritter
- // Just a very simple function
- //-----
- #include <iostream>
- using namespace std;
- //-----
- // Prototypes
- int Sum (int num1, int num2);
- //-----

```
•
•     int main ()
•     {
•         float num2 = 8;
•         float num1 = 22;
•
•             cout << num1 << " + " << num2 << " = " << Sum (num1,
•                                         num2)
•                         << endl << endl;
•
•             return 0;
•     }
•
• //-----
•
•     int Sum (int num1, int num2)
•     {
•         return num1 + num2;
•     }
•
• //-----
```

- // function2.cpp
- //
- // Simple program showing function use with a variable parameter
- //
- // Version
- // 001 Nicola Ritter
- // Just a very simple function
- //-----
- #include <iostream>
- using namespace std;
- //-----
- // The '&' below means that the value can be changed within the
- // procedure or function
- bool Divide (float num1, float num2, float &result);
- //-----

```
• int main ()  
• {  
•     float num1 = 0;  
•     float num2 = 0;  
•     float result = 0;  
•  
•     cout << "Enter two numbers: ";  
•     cin >> num1 >> num2;  
•  
•     if (Divide (num1, num2, result))  
•     {  
•         cout << num1 << " / " << num2 << " = " << result << endl;  
•     }  
•     else  
•     {  
•         cout << "You cannot divide by zero!" << endl;  
•     }  
•  
•     cout << endl;  
•     return 0;  
• }
```

- //-----
- **bool** Divide (**float** num1, **float** num2, **float** &result)
- {
- **if** (num2 == 0)
- {
- **return** false;
- }
- **else**
- {
- result = num1 / num2;
- **return** true;
- }
- }
- //-----

Procedures

- Procedures, in C++, are simply **void** functions:
-

```
void Sum (int num1, int num2, int &result)  
{  
    result = num1 + num2;  
}
```

- When it is called:

```
Sum (num1, num2, result);
```

Style and Layout

- Use a comment line `//-----` to separate sections of a program.
- Follow the placement of braces as shown in the notes and the book.
- Indent the same amount each time: preferably four spaces.
- Ideally, you should be able to see the whole of a function on the screen: if you cannot then you need to split the function up.
- Avoid having more than two ‘return’ statements in any function.
- Avoid having two ‘return’ statements that are physically far apart in a function.
- Braces must be used for *every* block, even if only one line long. [1]

Readings

- Textbook (by Malik): Chapters 4, 5, and chapter on User Defined Functions.



Murdoch
UNIVERSITY

Data Structures and Abstractions

Pointers

Lecture 5

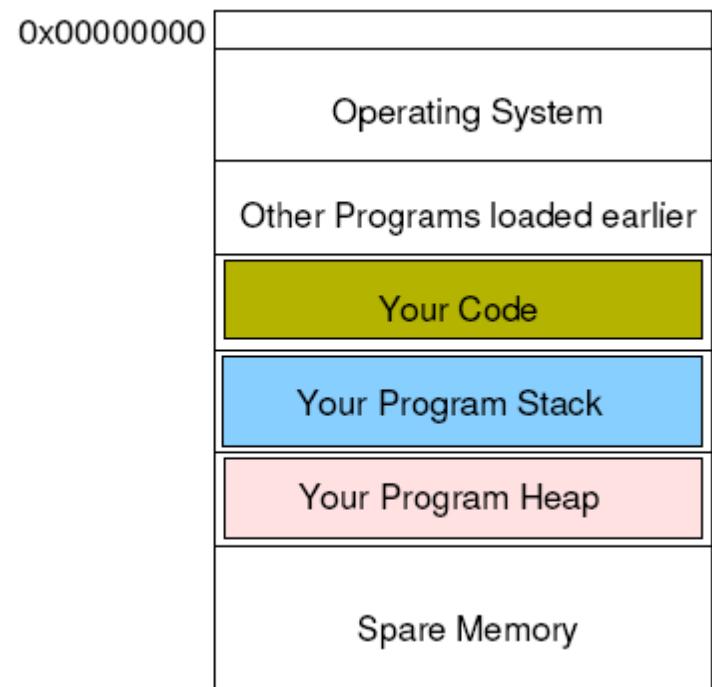


Error Types Reminder

- Don't forget the different error types when coding.
- Debugging is an absolute necessity when coding and even more so when coding pointers, so discussing this briefly here is appropriate.
- There are five types of common errors:
 - *Syntax* errors are those that prevent your code from compiling because of incorrect use of language grammar.
 - Semantic errors are errors in meaning. For example, if you have an apple object and you try to add to an orange object. Much more common ones are when you try to assign a float value to an integer variable. These can also be called type errors.
 - When your code compiles and your program runs as planned, but the output is incorrect, you have a *logic* error. **That is why you need a test plan! And why you actually run through it!** You had to use the test tables in the revision exercise.
 - When your program crashes it is a *run-time* error. **You need a test plan and testing.**
 - Finally if the input data is incorrect in some way, then your output will be incorrect also: a GIGO error.

RAM

- When your program runs, the OS allocates RAM for its use. [1]
- This RAM is divided up into different sections for use in different ways by your program. The layout shown below can be different for different architectures and OS. **Covered in your first year unit.**
- The program stack stores variables, parameters etc. that are defined when you write your program.
- The program heap is used for memory locations that are defined as your program runs: *dynamically* allocated variables.



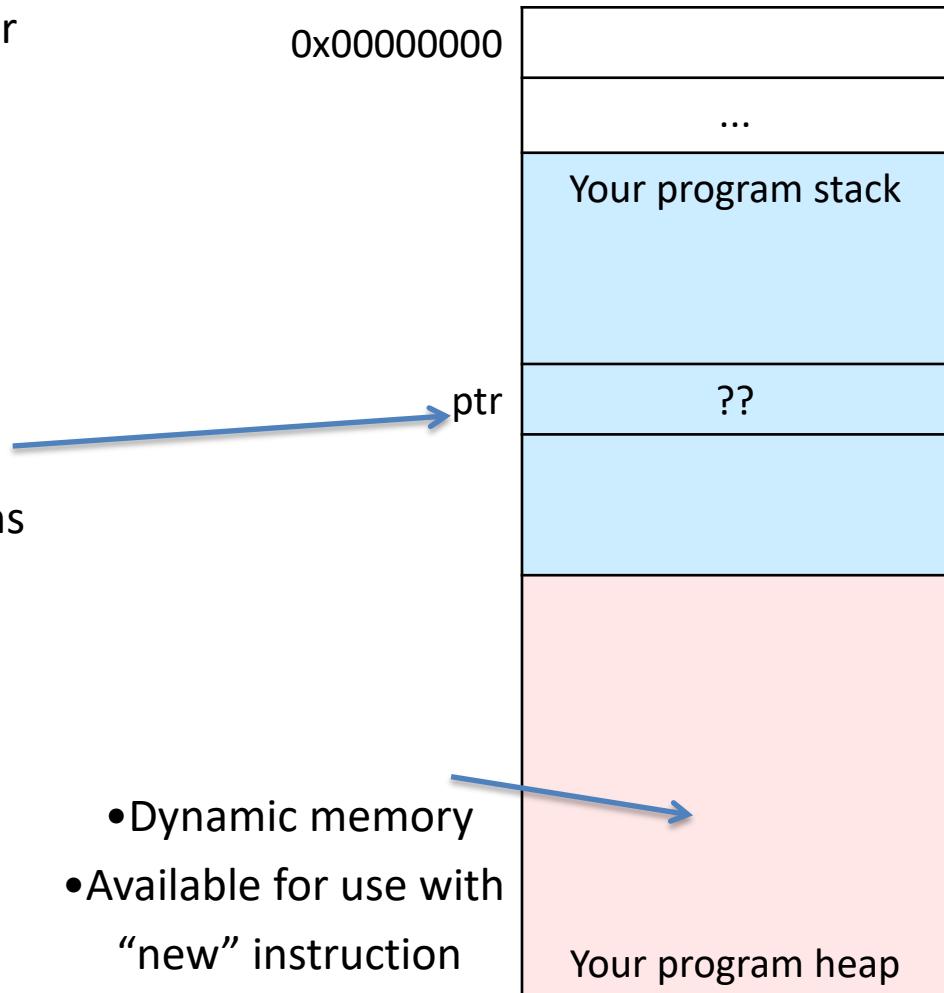
Pointers

- Pointers cause more heartache than is really necessary. [1]
- All they are is a variable that stores the address of another variable.
- Of course it is also possible to continue this ad-infinitum, which means that you can end up with an address, of an address, of an address...
- But in actual fact it is rare that you go beyond one, or two, levels of *dereference*.

Declaring Pointers

- This declares a pointer to an integer sized memory location.
- It does *not* allocate the memory location for you to store the actual data.
- Memory to store ptr itself but not the data. [1]
- Contents of ptr is whatever happens to be there by chance.

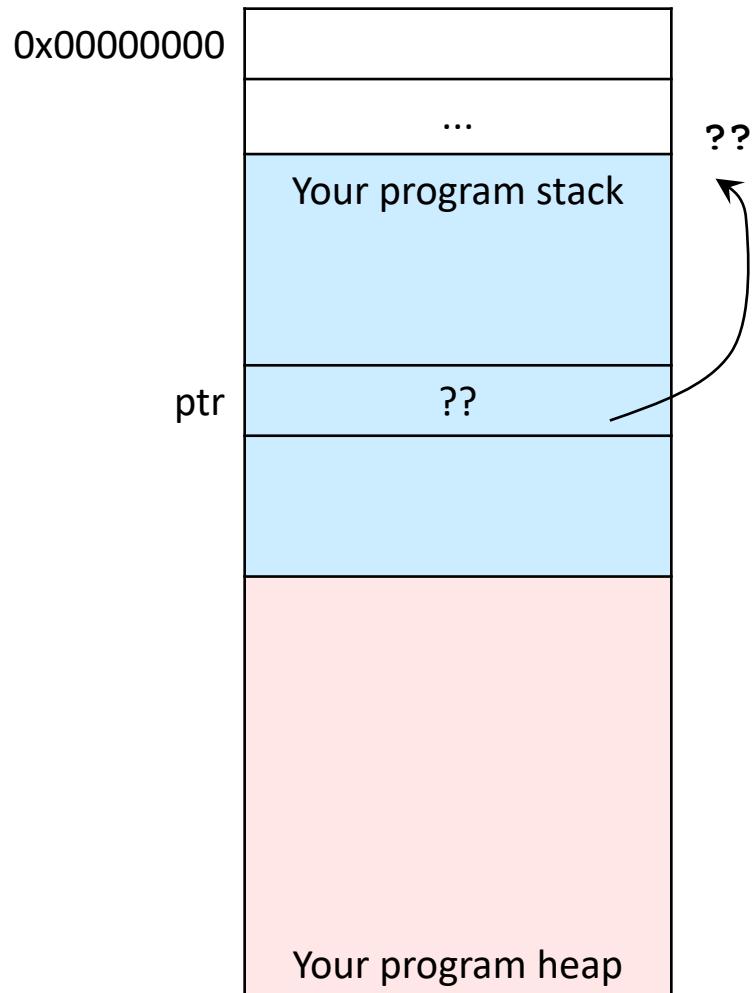
```
int f1 ()  
{ // local vars on stack  
    int *ptr;  
}
```



Declaring Pointers

- This means that trying to use the memory location contained in the pointer variable will crash the program or cause strange events.
- This is because the contents of `ptr` could be anything; i.e. it could be pointing *anywhere* in memory. [1]

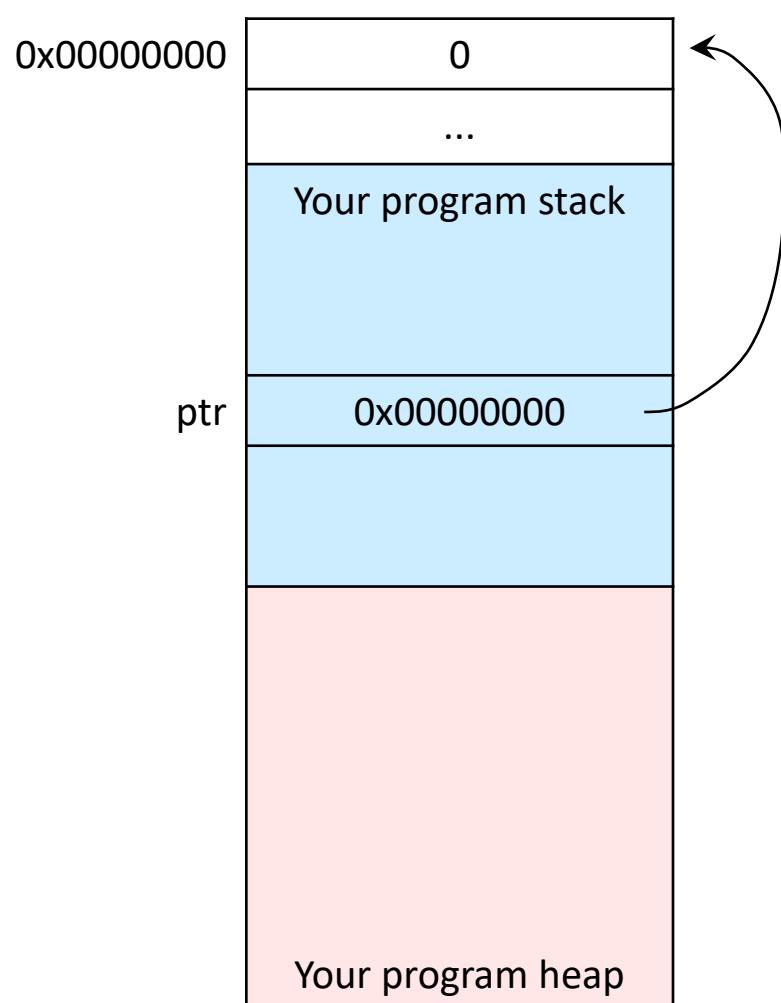
```
int *ptr;
```



Declaring Pointers Safely

- To prevent accidental alteration of anything important, pointers should always be **initialised** to NULL.
- The zeroth memory location of RAM is kept empty for this reason. [1]

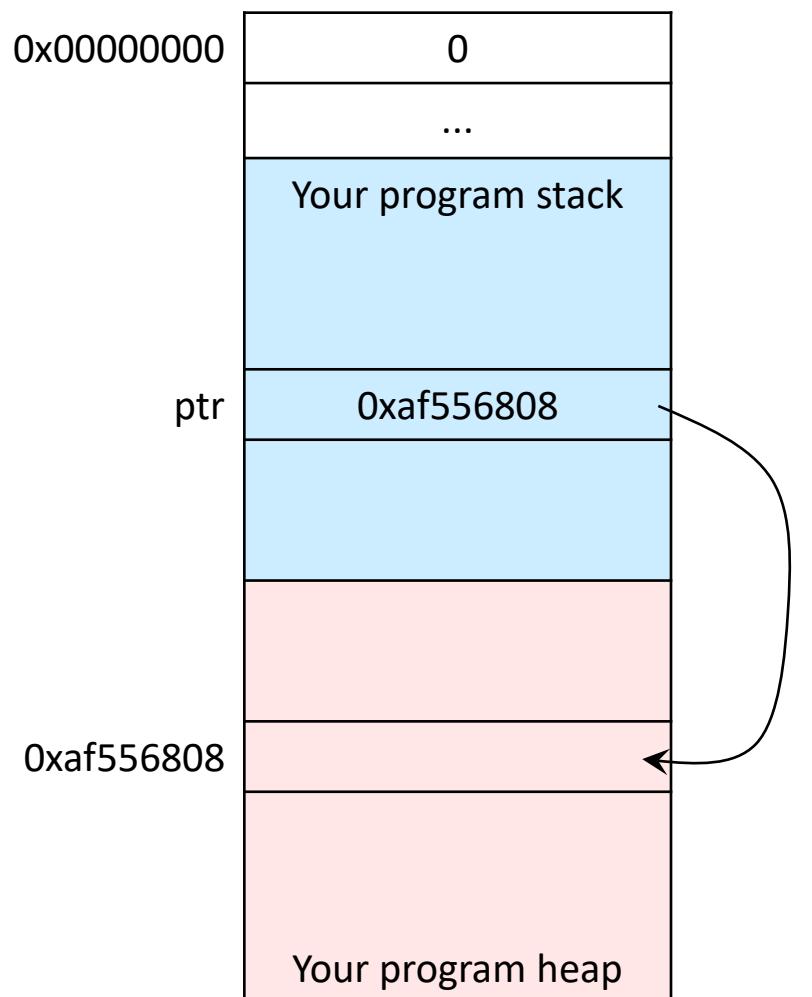
```
int *ptr = NULL;
```



Allocating Memory

- The allocation of memory is then done with the `new` keyword.
- This allocates a memory location on the heap of the given size (in this case an int).

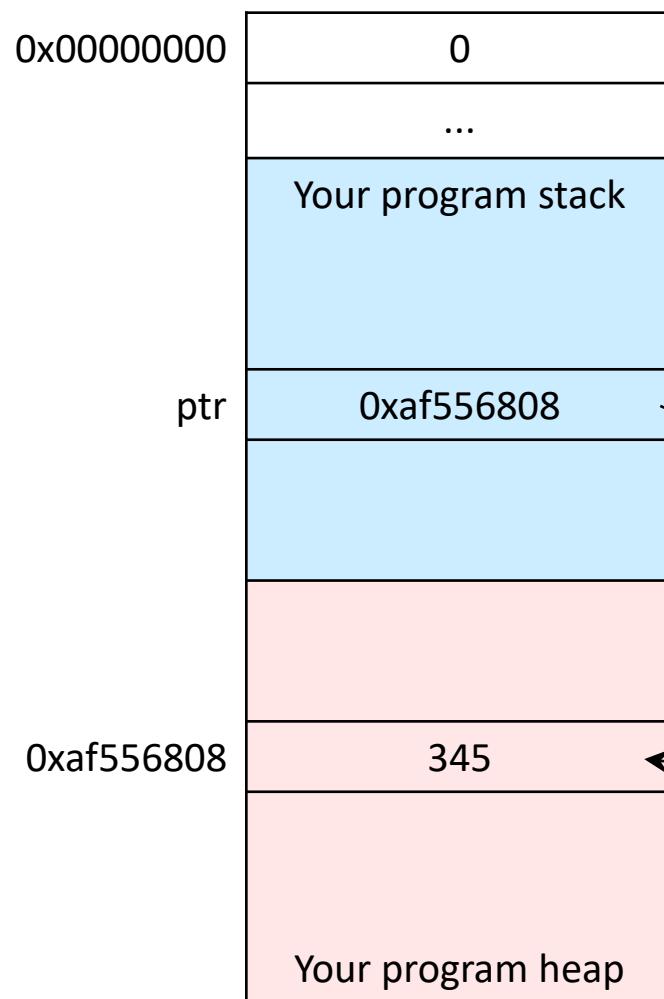
```
int *ptr = NULL;  
  
ptr = new int; [1]
```



Using Pointers

- To place a value in the memory location, the `*` dereferencing operator is used. [1]

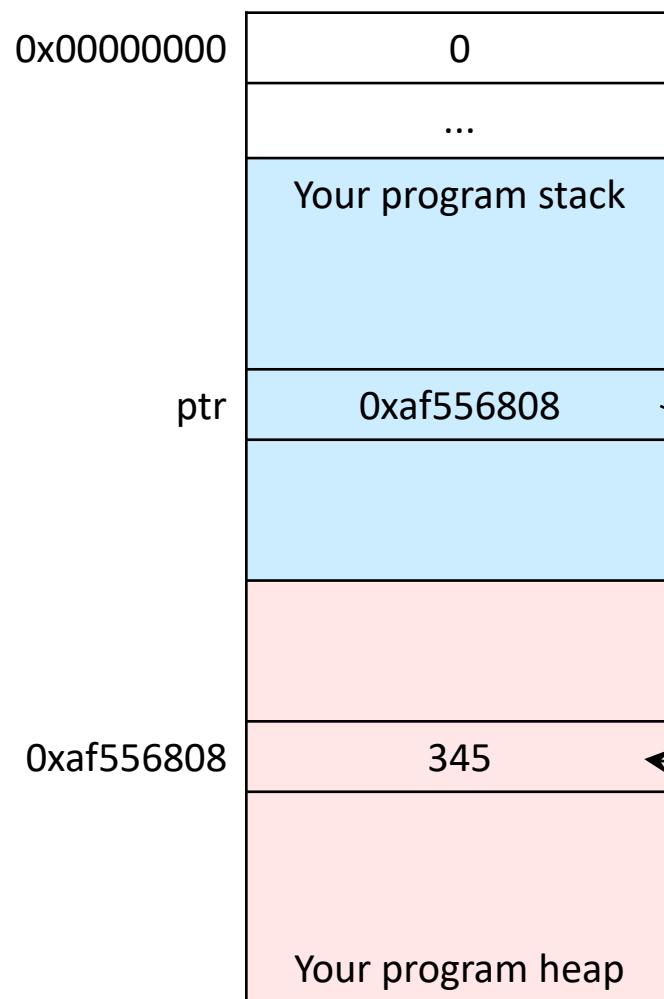
```
int *ptr = NULL;  
  
ptr = new int;  
  
*ptr = 345;
```



Using Pointers

- This is also used for accessing the location for output etc.

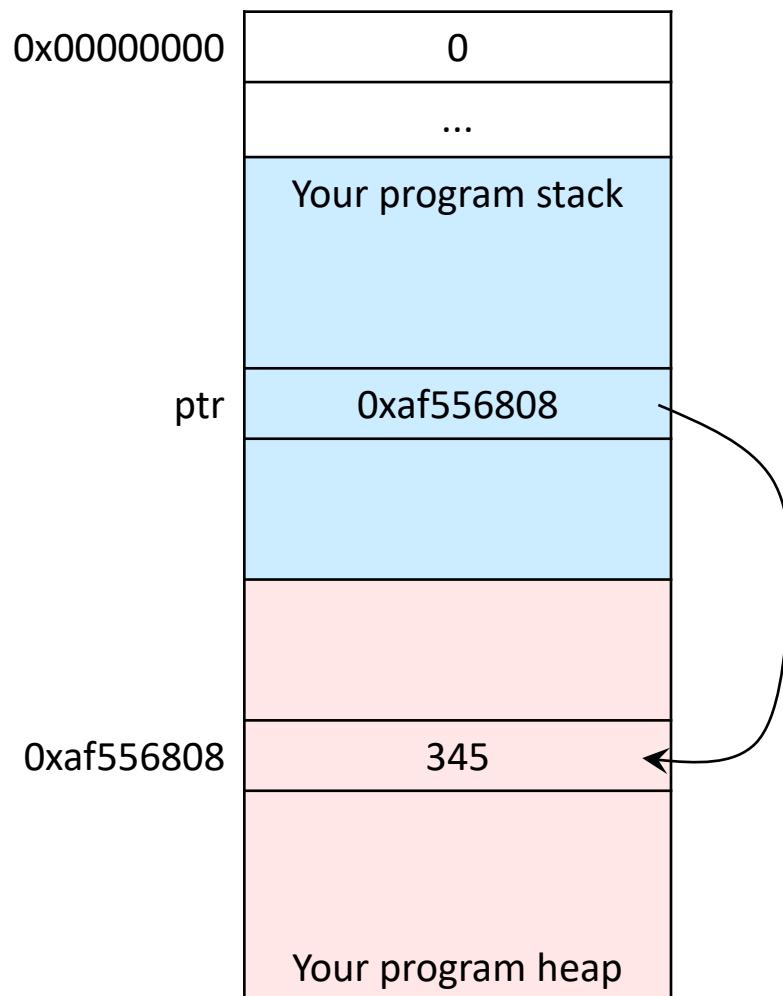
```
int *ptr = NULL;  
  
ptr = new int;  
  
*ptr = 345;  
  
cout << *ptr << endl;
```



Where is the Pointer Pointing?

- You could also output the location of the memory being used, rather than the contents of the memory location.

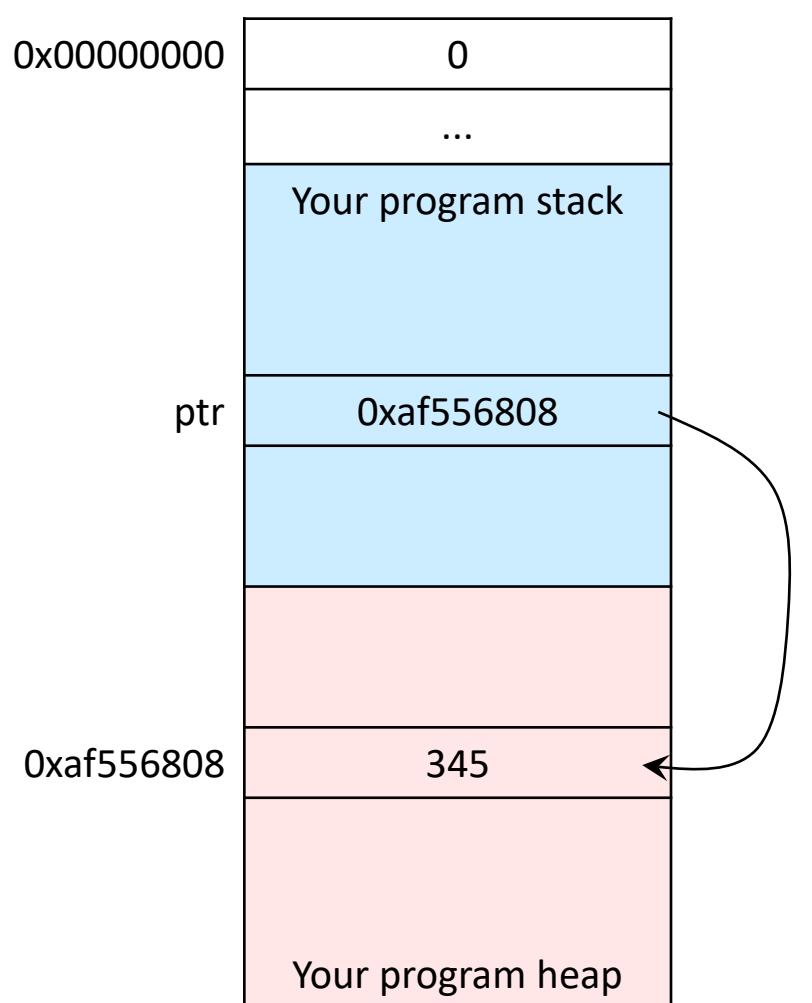
```
int *ptr = NULL;  
  
ptr = new int;  
  
*ptr = 345;  
  
cout << ptr << endl;
```



Releasing Memory

- Every **new** must have a matching **delete**, so that memory is released back to the OS.

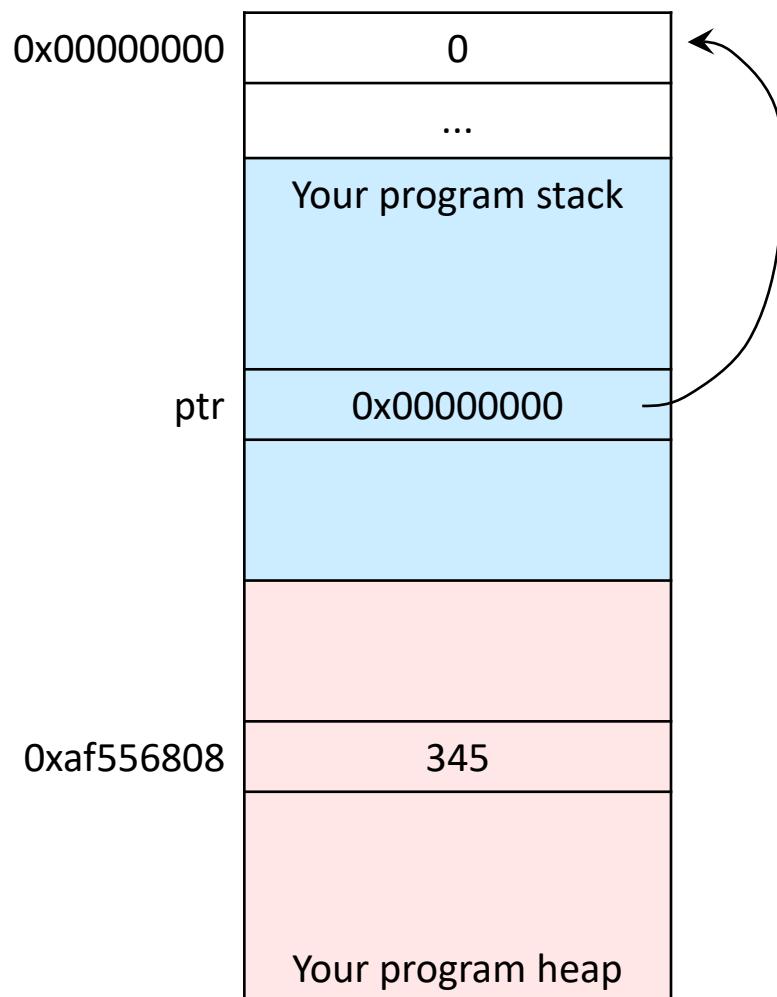
```
int *ptr = NULL;  
  
ptr = new int;  
  
...  
  
delete ptr;
```



Releasing Memory Safely

- Followed, of course, by reassigning the pointer to NULL, so that it does not point to memory over which it no longer should have control.

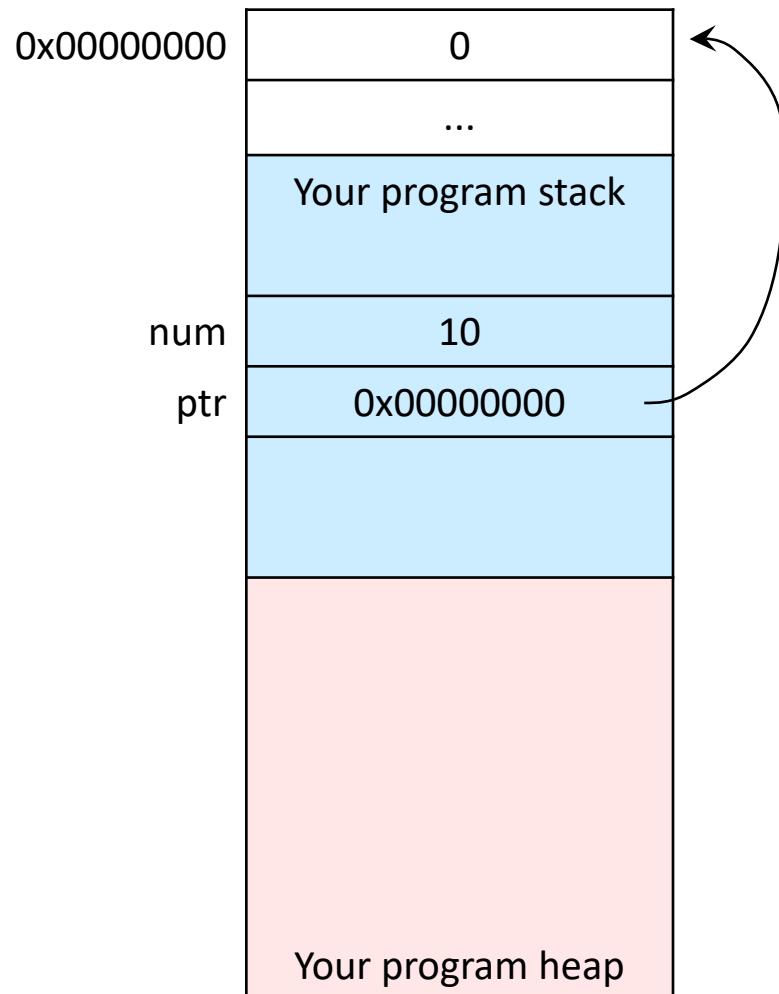
```
int *ptr = NULL;  
  
ptr = new int;  
  
...  
  
delete ptr;  
  
ptr = NULL; // to be safe
```



Pointing at Other Variables

- Pointers can also point at other variables:

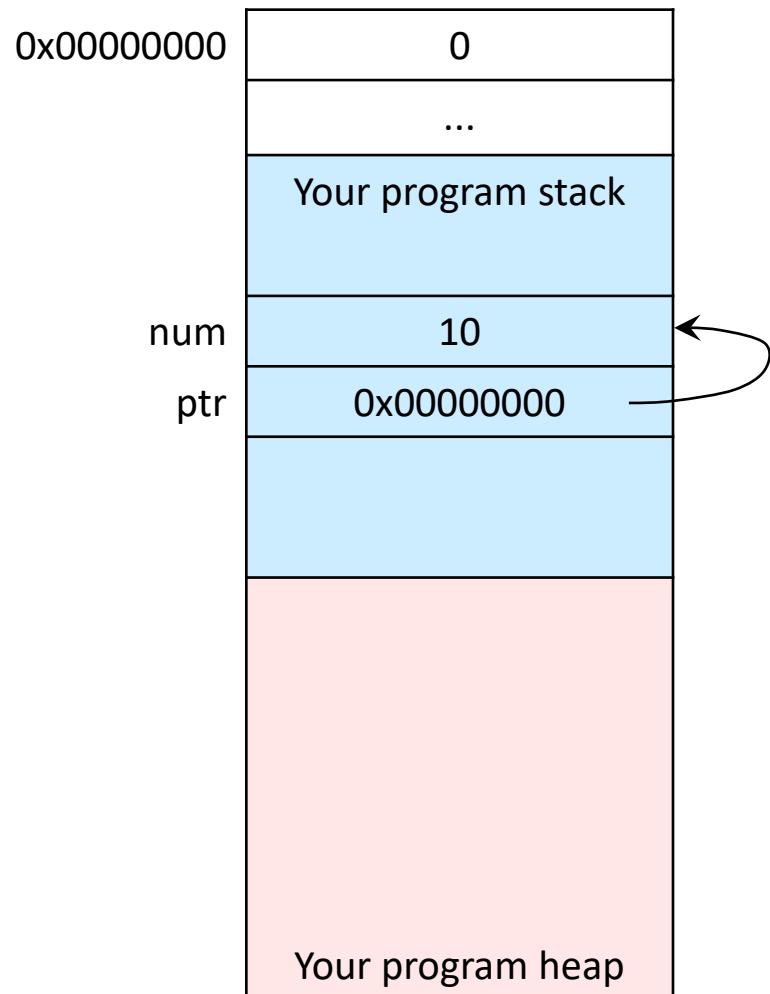
```
int *ptr = NULL;  
int num = 10;
```



Pointing at Other Variables

- Pointers can also point at other variables.

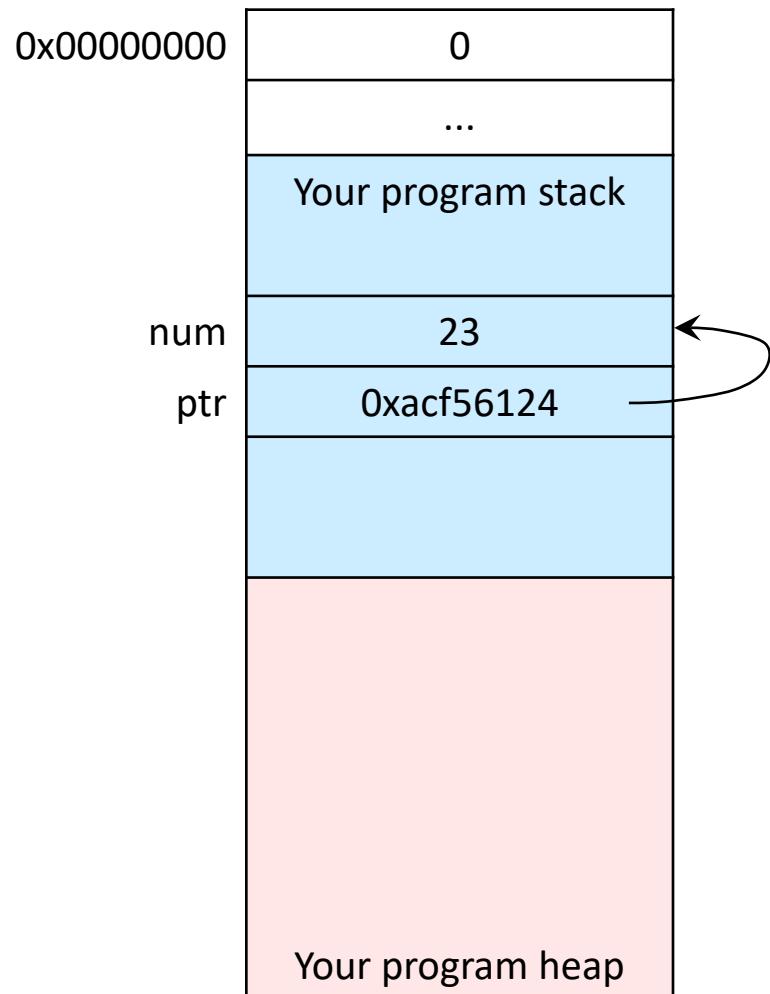
```
int *ptr = NULL;  
  
int num = 10;  
  
ptr = &num;
```



Pointing at Other Variables

- And change their value.

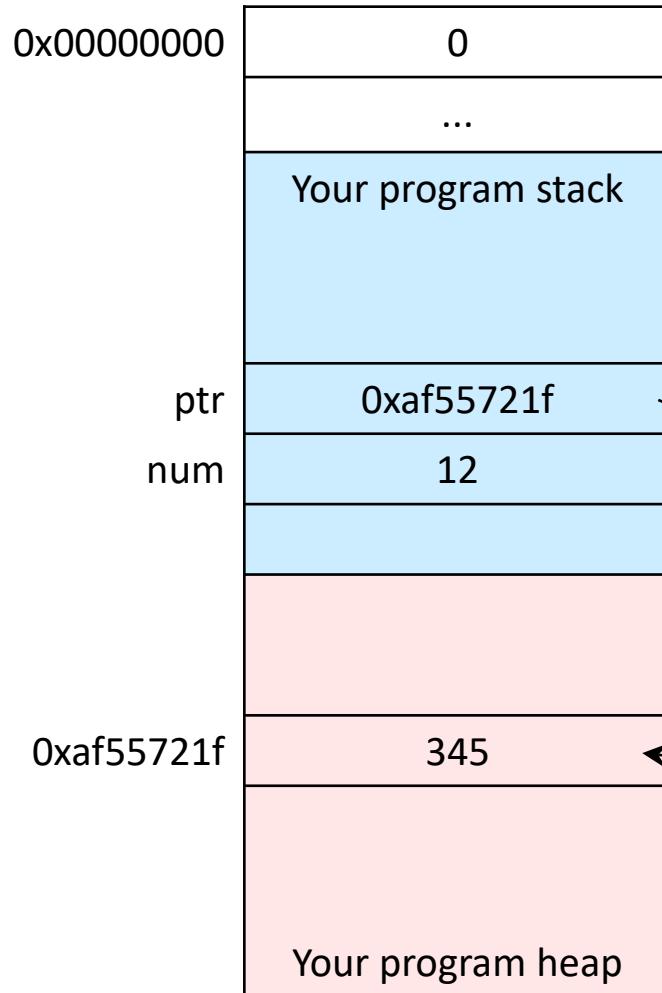
```
int *ptr = NULL;  
  
int num = 10;  
  
ptr = &num;  
  
*ptr = 23;
```



Memory Leaks

- However care must be taken not to cause a memory leak.

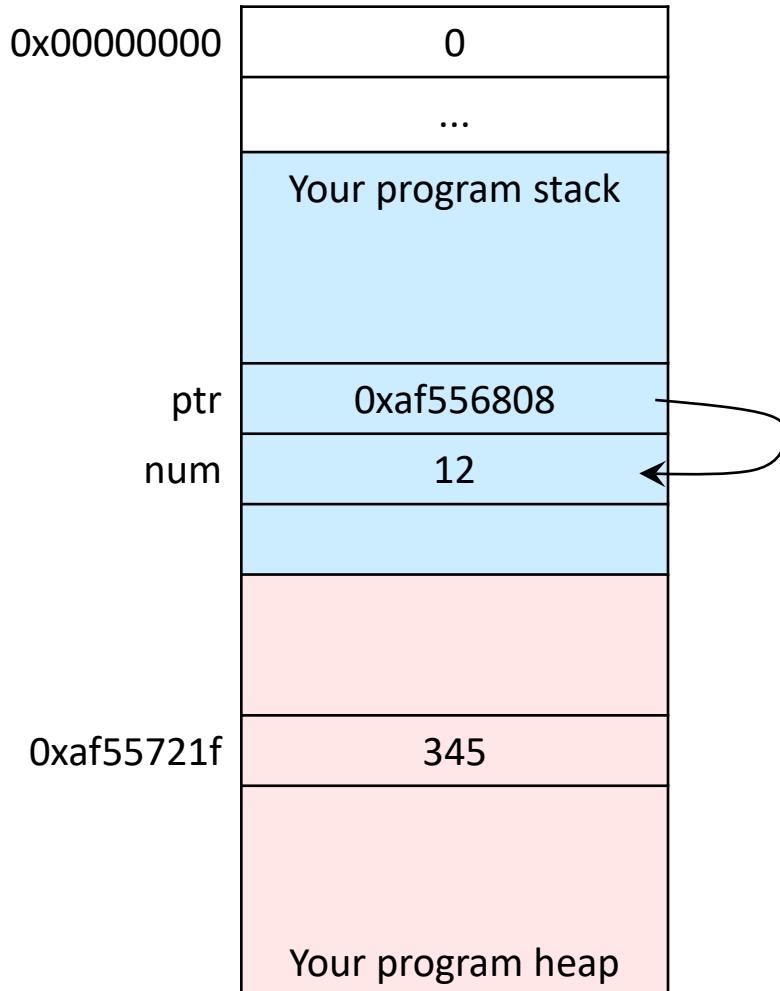
```
int num = 12;  
  
int *ptr = new int;  
  
*ptr = 345;
```



Memory Leaks

- However care must be taken not to cause a memory leak.

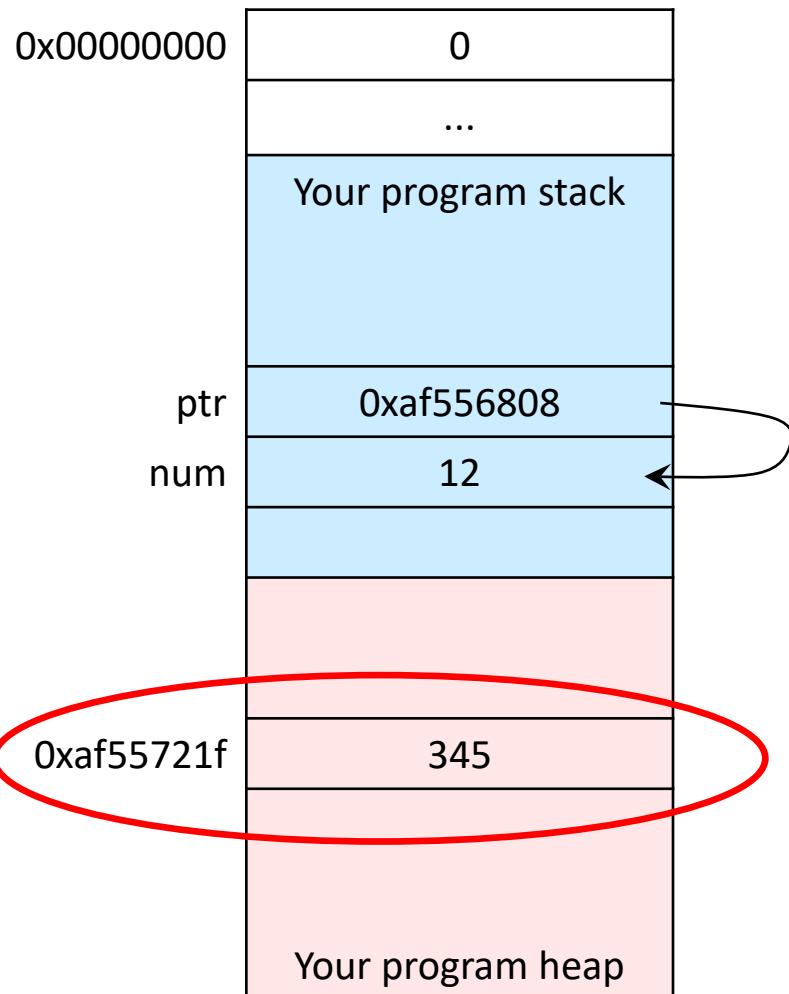
```
int num = 12;  
  
int *ptr = new int;  
  
*ptr = 345;  
  
ptr = &num;
```



Memory Leaks

- The circled memory location is 'lost' until the program ends.

```
int num = 12;  
  
int *ptr = new int;  
*ptr = 345;  
  
ptr = &num;
```



Avoiding Memory Leaks

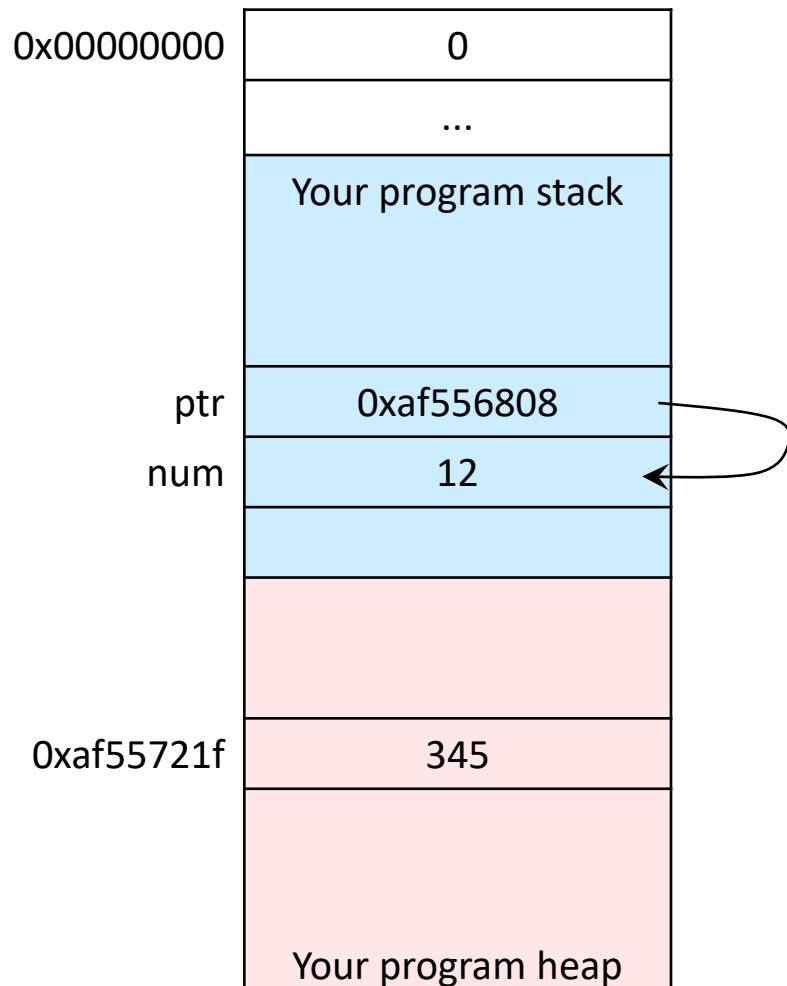
- Of course, there should have been a `delete` between the last two lines of code.
- This releases the memory back to the OS again.

```
int num = 12;

int *ptr = new int;
*ptr = 345;

delete ptr;

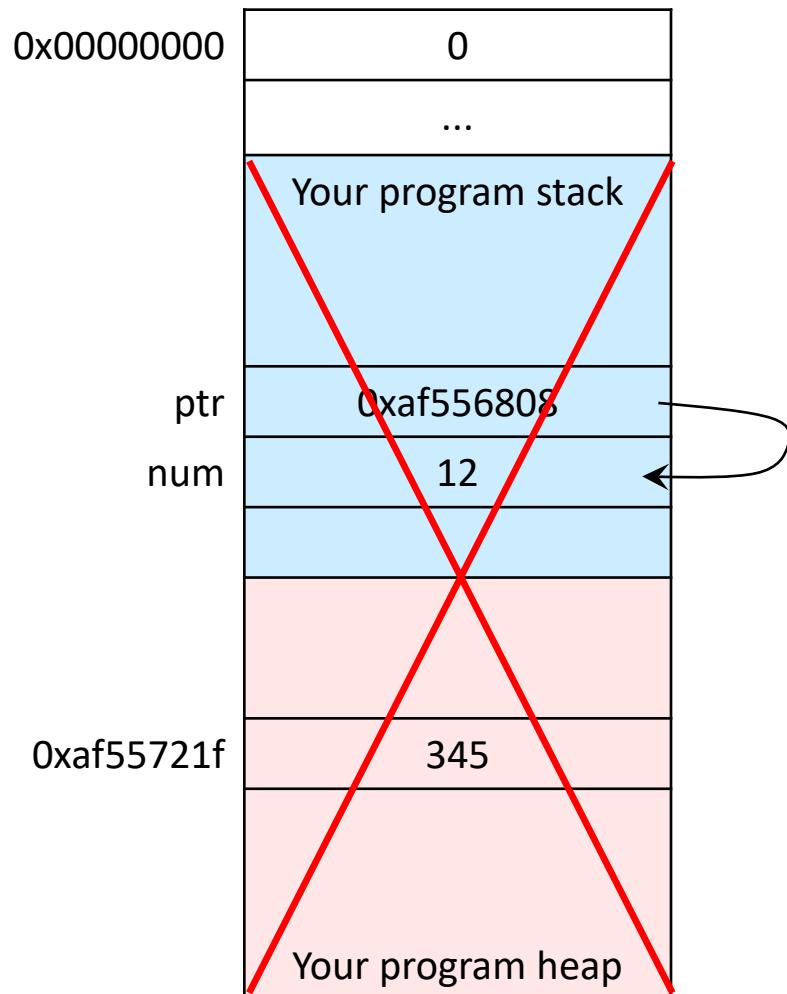
ptr = &num;
```



Care with delete

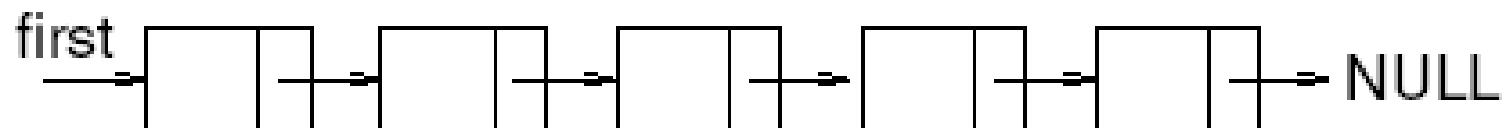
- But beware where you put it: trying to delete stack memory will cause trouble.

```
int num = 12;  
  
int *ptr = new int;  
  
*ptr = 345;  
  
ptr = &num; //num not on heap  
  
delete ptr; // oops
```



Uses of Pointers

- There are two main uses for pointers.
- The first is for array and string access.
- The second is where pointers are used to create lists or trees: data structures where the next piece of data can only be found by traversing a link from the last piece:



Malik: Chapter on Pointer - exercises

- 2. Given the declaration:
 - `int x;`
 - `int *p;`
 - `int *q;`
- Mark the following statements as valid or invalid.
- If a statement is invalid, explain why:
 - a. `p = q;`
 - b. `*p = 56;`
 - c. `p = x;`
 - d. `*p = *q;`
 - e. `q = &x;`
 - f. `*p = q;`

- 3. What is the output of the following C++ code?
- `int x;`
- `int y;`
- `int *p = &x;`
- `int *q = &y;`
- `*p = 35;`
- `*q = 98;`
- `*p = *q;`
- `cout << x << " " << y << endl;`
- `cout << *p << " " << *q << endl;`

- 4. What is the output of the following C++ code?

- `int x;`
- `int y;`
- `int *p = &x;`
- `int *q = &y;`
- `x = 35;`
- `y = 46;`
- `p = q;`
- `*p = 18;`
- `cout << x << " " << y << endl;`
- `cout << *p << " " << *q << endl;`

Readings

- Textbook (by Malik): Chapter on Pointers, Classes, .. etc. See subsections on Pointer Data Type and Pointer Variables; Address of operator; Dereferencing operator. A different edition may have a different chapter number and pages.
- 101 Coding Standards: **Rules 51 and 52**. It is one of the important references (see unit outline) we use in the unit. See [101 C++ Coding Standards online resource. \[1\]](#)
- Watch the videos on pointers “Video Lecture on Pointers.htm” Function pointers are covered when we cover the tree data structure later on.
- Find out about “RAII”. Why is the RAII concept so important that you should not violate it?

Videos

- Pointers - Stanford University
<https://www.youtube.com/watch?v=H4MQXBF6FN4>
- Bits and bytes; floating point representation - Stanford University
<https://www.youtube.com/watch?v=jTSvthW34GU>
- How pointers get used; usage of void pointers
<https://www.youtube.com/watch?v= eR4rxnM7Lc>



Murdoch
UNIVERSITY

Data Structures and Abstractions

Parameters

Lecture 6



Parameters

- Parameters can be passed in four main ways.
 - by value
 - by reference
 - by constant reference
 - by pointer

Value Parameters

```
void Func1 (int number);  
...  
int num = 8;
```

number is a value parameter

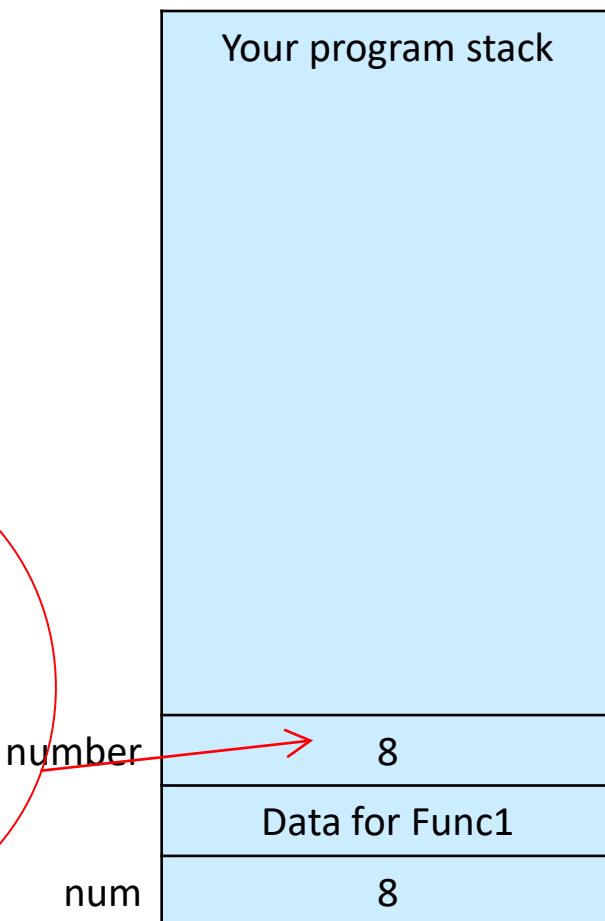
num 8

Your program stack

Value Parameters

```
void Func1 (int number);  
...  
  
int num = 8;  
Func1 (num);
```

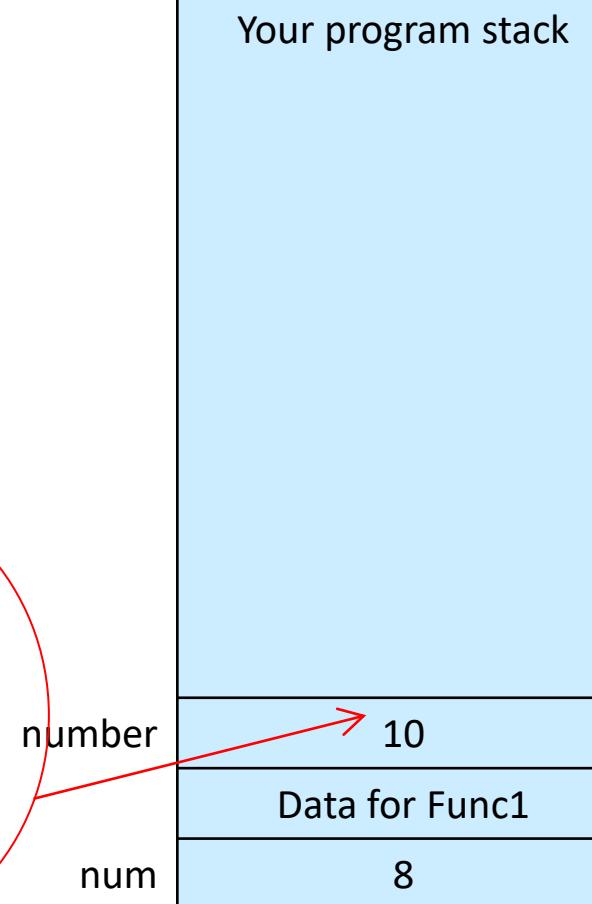
number is given
an initial value
from the variable
num



Value Parameters

```
void Func1 (int number);  
...  
  
int num = 8;  
Func1 (num);
```

number is
changed to 10
within the Func1
function.



Value Parameters

Your program stack

```
void Func1 (int number);  
...  
  
int num = 8;  
Func1 (num);
```

But **num** remains 8
after the function
has completed

num → 8

Reference Parameters

```
void Func2 (int &number); // [1]
```

```
...
```

```
int num = 8;
```

number is a
reference
parameter i.e.
another name for
something

num

8

Your program stack

Reference Parameters

```
void Func2 (int &number);  
...  
  
int num = 8;  
Func2 (num);
```

number is just
another reference
to (name for) the
location also called
num

number

num

Your program stack

Data for Func2

8

Reference Parameters

```
void Func2 (int &number){  
    number = 10; //changes number to 10  
}  
  
-----  
int num = 8;  
Func2 (num);
```

when **number** is changed it changes the value of **num**, because they are really the same thing.

number

num



Reference Parameters

```
void Func2 (int &number);  
...  
  
int num = 8;  
Func2 (num);
```

Your program stack

num remains 10
after the function
has completed

num 10

Constant Reference Parameters

```
void Func3 (const int &number);  
...  
int num = 8;
```

number is a
constant reference
parameter

Your program stack

num 8

Constant Reference Parameters

```
void Func3 (const int &number);  
...  
  
int num = 8;  
Func3 (num);
```

number refers to the location also called num, but number is locked as a constant while Func3 is running

number

num

Your program stack

Data for Func3

(const) 8

Constant Reference Parameters

```
void Func3 (const int &number);  
...// attempt to change number  
...// will not compile - good  
  
int num = 8;  
Func3 (num);
```

If **Func3** tries to alter **number**, the program *will not compile!*

number

num

Your program stack

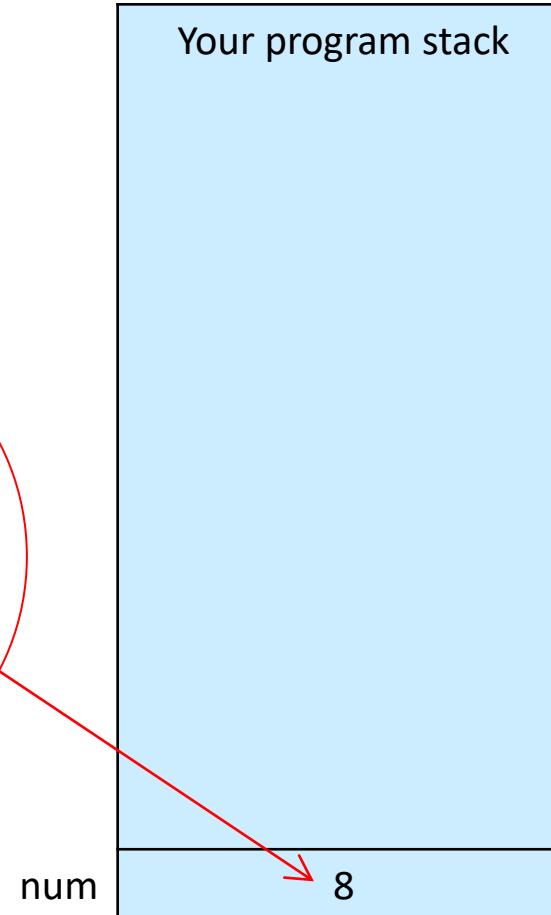
Data for Func3

(const) 8

Constant Reference Parameters

```
void Func3 (const int &number);  
...  
  
int num = 8;  
Func3 (num);
```

Therefore **num** will remain as 8 as the function can't run (wouldn't compile)



Pointer Parameters

```
void Func4 (int *ptr);  
...  
int num = 8;
```

ptr is a pointer parameter, therefore within **Func4**, **ptr** is a pointer *not* an integer

Your program stack

num 8

Pointer Parameters

```
void Func4 (int *ptr);  
...  
  
int num = 8;  
Func4 (&num);
```

ptr stores the
address of num

*ptr

num

ptr

&num

Data for Func4

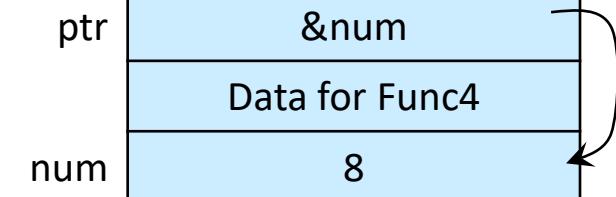
8

Your program stack

Pointer Parameters

```
void Func4 (int *ptr);  
...  
  
int num = 8;  
Func4 (&num);
```

Therefore ***ptr**
becomes a
reference to **num**



Pointer Parameters

```
void Func4 (int *ptr);  
...  
  
int num = 8;  
Func4 (&num);
```

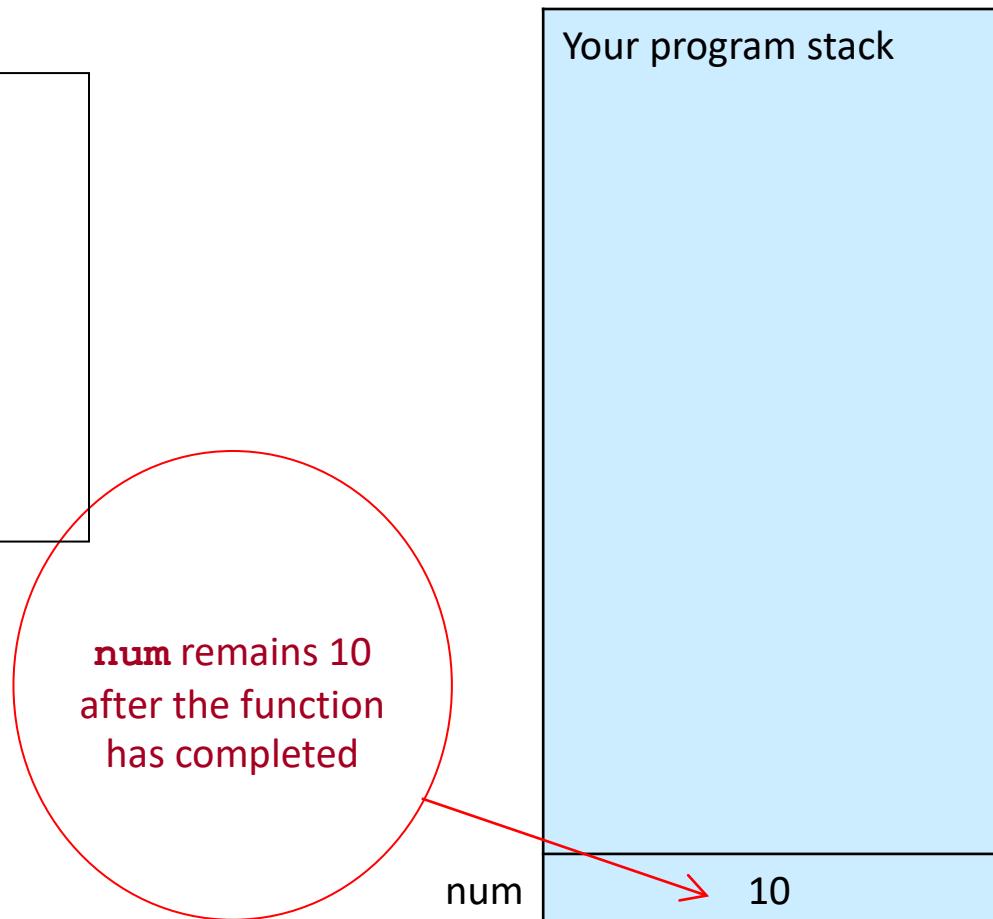
If **Func4** sets
***ptr** to 10, then
num is changed to
10



Pointer Parameters

```
void Func4 (int *ptr);  
...  
  
int num = 8;  
Func4 (&num);
```

- For further work (not necessary for this unit), find out about smart pointers, auto_ptr, and Opaque pointer. [1]



Pointers and References

- Find out about the following:
- It is possible to declare a pointer with no initial value? Is it possible to declare a reference which does not contain an initial value?
- A pointer variable can be changed to point to something else. Can this be done with a reference?
- A pointer can be set to contain the NULL (or `nullptr`) value. Can you make a reference NULL (or `nullptr`)? [1]
- Can you do pointer like arithmetic on references?

Readings

- Chapter(s) on User-Defined Functions, section on Value Returning Functions; section on Reference variables as parameters.
- If you are using another edition of the textbook, look up the chapter title and section number in the contents page.



Murdoch
UNIVERSITY

Data Structures and Abstractions

Arrays & Records

Lecture 7



- The revision exercise must be completed before starting on Topic 3.
 - A revision of arrays from the revision exercise is needed for this topic.

Arrays in C++

- Arrays in C++ are not a predefined type, they have to be defined by the programmer: [1]

```
const int SIZE = 10; [2]  
...  
typedef int ArrayType[SIZE];  
...  
ArrayType numbers;
```

- This declares an array of 10 integer point numbers with indexes from 0 to 9.
- Processing of arrays is the same in C++ as in C or Java, with [] giving access to elements.

Within Memory

- Within memory, the compiler has generated code to allocate 10 contiguous slots on the stack, each large enough to store one floating point number.
- The variable '**numbers**', is in fact a “*pointer*” to the first of the 10 memory locations. But you cannot put a new address into it. What are the consequences if you were able to do this?
- This can be seen if you output the array variable without an index:
`cout << hex << numbers << endl`
would give output something like: **0x23ab34 [1]**
which is the address in RAM of the 0th location out of the 10 integers.
- However,
`cout << numbers[0] << endl;`
will output the contents of the 0th memory location.

- // Arrays1.cpp
- // Demonstrating array use
- // Version: 01, Nicola Ritter
- // Version: 02, SMR
- //-----
- #include <iostream>
- using namespace std;
- //-----
- const int SIZE = 10;
- const int END = -1;
- //-----
- // Define a new type that is an array of SIZE integers
- typedef int IntArray[SIZE];

- //-----
- // Returns the number of input values
- //This is called a forward declaration
- // Both the lines shown next are examples of forward declaration
- int Input (IntArray &array);
- void Output (int size, const IntArray &array);
- // Why const?
- //-----
- int main()
 - {
 - IntArray array;
 - int size = Input (array); // size returns how many
 - Output (size, array); // this is a procedure
 - return 0;
 - }
- //-----

- ```
int Input (IntArray &array)
{
 cout << "Enter up to " << SIZE
 << " positive integers, pressing <Enter> after each integer"
 << endl << "Use " << END << " to end input."
 << endl;
```
- ```
int num;
int index;
cout << "Enter integer: ";
cin >> num;
for (index = 0; index < SIZE && num != END; index++)
{ // is the for loop the correct loop to use for this situation?
    array[index] = num;
    cout << "Enter integer: ";
    cin >> num;
}
return index;
}
```

- //-----
- **void Output (int size, const IntArray &array)**
- {
- cout << endl << "Array at address " << array << " contains:"
- << endl;
- **for (int index = 0; index < size; index++)**
- {
- cout << index << ") " << array[index]
- << " at memory location " << &(array[index])
- << endl;
- }
- cout << endl;
- }
- //-----

Two Dimensional Arrays

- Arrays of more than one dimension are defined in a similar way:

float

twoDimArray [ROWS] [COLS] ;

- Access is as per normal:

twoDimArray [row] [col] ;

Dynamic Arrays

- Arrays can be declared dynamically as well:

```
int *array = new int[size];
```

declares an array of 'size' integers.

- As it is a dynamic declaration, the integers are on the *heap* rather than the stack. Don't forget to delete them afterwards when you no longer need them.
- If there is not enough memory (new fails), array is set to NULL (or the preferred **nullptr**).
- However use of the array does not change unless it was set to NULL. What would happen if it was set to NULL?

- // TwoDimArray.cpp
- // Program designed to show the declaration and use of a dynamically
- // created two dimensional array
- //
- // Version
- // 001 First Attempt, Nicola Ritter
- // modified smr
- //-----
- #include <iostream>
- #include <iomanip>
- using namespace std;
- //-----
- const int COL_WIDTH = 14;
- //-----Forward declarations-----
- void GetSizes (int &rows, int &cols);
- float **CreateArray (int rows, int cols);
- void Output (int rows, int cols, float **array);
- void Initialise (int rows, int cols, float **array);
- void Delete (int rows, float **&array);
- //-----

```
• int main ()
• {
•     // Declare a pointer to an array of pointers
•     float **array = NULL;
•     int rows, cols;
•
•     GetSizes (rows, cols);
•
•     // Get memory for the 2 dim array
•     array = CreateArray (rows, cols);
•
•     if (array != NULL)
•     {
•         // Set the table to contain 0s
•         Initialise (rows, cols, array);
•
•         // Output to check
•         Output (rows, cols, array);
•
•         Delete (rows, array); // this is a user defined delete, not C++ delete [1]
•     }
•
•     // put in an else part which says that memory for array could not be created.
•
•     return 0;
• }
```

- //-----
- **void GetSizes (int &rows, int &cols)**
- {
- cout << "How many rows do you want? ";
- cin >> rows;
- cout << "How many columns do you want? ";
- cin >> cols;
- cout << endl;
- }
- //-----

```
•     float **CreateArray (int rows, int cols)
•     {
•         // Get an array of pointers for the rows
•         float **array = new float* [rows]; //draw this structure
•
•         if (array != NULL) // how about an else
•         {
•             // Now get a row for each of these pointers
•             for (int index = 0; index < rows; index++)
•             {
•                 array[index] = new float[cols];
•             }
•
•             // Check that allocation occurred
•             if (array[rows-1] == NULL)
•             {
•                 Delete (rows, array);
•             }
•         }
•
•         return array;
•     }
```

- //-----
- void Output (int rows, int cols, float **array)
- {
- cout << "The array values are:" << endl;
- for (int row = 0; row < rows; row++)
- {
- for (int col = 0; col < cols; col++)
- {
- cout << setw(COL_WIDTH) << array[row][col];
- }
- cout << endl;
- }
- cout << endl;
- }
- //-----

- **void Initialise (int rows, int cols, float **array)**
- {
- **for (int row = 0; row < rows; row++)**
- {
- **for (int col = 0; col < cols; col++)**
- {
- array[row][col] = 0;
- }
- }
- }
- **//-----**

- `void Delete (int rows, float** &array)`
- `{`
- `for (int row = 0; row < rows; row++)`
- `{`
- `if (array[row] != NULL)`
- `{`
- `delete [] array[row]; // draw a diagram`
- `}`
- `}`
- `delete [] array;`
- `array = NULL;`
- `}`
- `//-----`

Input Testing

- When testing a program it can become onerous to keep having to type in data.
- For this reason it is common to store test data in a file and **run the program from the command line**, redirecting data into the program:

```
aprogram.exe < data.txt
```

- The data stored in **data.txt** will be read into the program *as if it had been entered from the keyboard.* [1]
- Note that this is **not the same as file input**, it is a way to replicate keyboard typing, without having to do it again and again.
- Note that the output data can also be redirected to file:

```
aprogram.exe < data.txt > data.out
```

- // redirect.cpp
- // Testing redirected input from a file
- #include <iostream>
- using namespace std;
- int main ()
 - {
 - int number;
 - do
 - {
 - cin >> number;
 - cout << number << endl;
 - } [1] while (cin.good());
 - return 0;
- }

Stops the loop
when the
redirected file
reaches eof

Input File (data.txt)

2
45
67
8
912

User types in [2]

redirect < data.txt

Screen Output

2
45
67
8
912

Records

- Records (grouped data) are produced in C and hence C++ using the `struct` type:

```
typedef struct Person
{
    char firstName [SIZE];
    char surname [SIZE];
    int age;
};
```

- A variable of this type is then declared in the same way as any other type:

```
Person person; [1]
```

- Access to parts of the person is done using the dot operator:

```
cout << person.firstName << " "
    << person.surname << " "
    << person.age << endl;
```

An Array of Records

- Similarly, we could declare an array of records as:

```
Person people[ARRAY_SIZE];
```

- And access an individual part of a single record:

```
people[index].firstname
```

Readings

- Textbook by Malik
 - Chapter on Arrays and Strings
 - Chapter on Records



Murdoch
UNIVERSITY

Data Structures and Abstractions

C Character Functions and Strings

Lecture 8



C Character Functions [1]

- There are many useful character functions available in C++.
- They are actually standard C functions.
- To use them we must include the header file `<cctype>`.
- Note that the original C header file was `<ctype.h>`: within C++ the “.h”: is removed and a “c” added in front of the file name.
- Most of these functions query the classification of the character.
- Because they are C functions, not C++ functions, they return 0 for false and 1 for true.

Useful Character Functions

isalpha (ch)	Is it an alphabetic character?
isascii (ch)	Is it an ASCII character?
isblank (ch)	Is it a blank character?
isdigit (ch)	Is it a digit (0..9)?
islower (ch)	Is it a lowercase alphabetic character?
isupper (ch)	Is it an uppercase alphabetic character?
ispunct (ch)	Is it punctuation? In this sense punctuation is defined as any printable character that is not a space or an alphanumeric character.
isspace (ch)	Is it a space?
isxdigit (ch)	Is it a hexadecimal digit (a..f, A..F, 0..9)?

Useful Character Functions cont.

There are two other useful functions:

toupper (ch)	If 'ch' is a lowercase character, return the uppercase character, or else return it unchanged.	char ch = 'a' ; ch = toupper (ch) ;
tolower (ch)	If 'ch' is an uppercase character, return the lowercase character, or else return it unchanged.	char ch = '?' ; ch = tolower (ch) ;

C Strings

- C strings are simply character arrays, and therefore defined the same way as for other arrays.
- However, C strings must have a NULL character at their end, therefore if you want 20 characters, you must allow space for 21:

```
const int SIZE = 20;  
...  
typedef char StringType[SIZE+1];  
...  
StringType str;
```

- If you overflow your allocated space, strange things can happen. [1]

NULL

- The NULL character is the character with ASCII value 0:

```
char ch = 0;
```

- Or it can be designated as a character using a backslash:

```
char ch = '\0';
```

- In fact many control and formatting characters have backslash equivalents, the most useful being:

newline	'\n'
tab	'\t'
quotes	'\"'
backslash	'\\'

Useful C String Functions

- C (and hence C++) has one of the most powerful sets of string manipulation functions available.
- C String functions require the **<cstring>** header file to be included.
- The string functions *do no overflow checking!*
- The **require null terminated char arrays.**

String Function List

int strlen (str)	Returns the number of characters from the start of the string to the NULL character.
int strcmp(str1,str2) [1]	Returns 0 if they are identical, a negative number if str1 is <i>lexographically</i> less than str2, and a positive number if str2 is greater than str1.
int strncmp(str1,str2,n)	Compares the first 'n' characters of each string. Returns 0 if they are identical, a negative number if str1 is <i>lexographically</i> less than str2, and a positive number if str2 is greater than str1.
int strcpy(dest,src) [1]	Copies src to dest, but does no bounds check.
int strncpy(dest,src,n)	Copies the first 'n' characters from src to dest, does no bounds check.

String Function List Continued

<code>int strcat (dest, src)</code>	Copies src to the end of dest, does no bounds check. [1]
<code>int strncat (dest, src, n)</code>	Copies the first 'n' characters of src to the end of dest, does no bounds check.
<code>char* strchr (str, ch)</code>	Searches str for the first occurrence of ch. Returns a <i>pointer</i> to ch, or NULL if the character does not occur.
<code>char* strrchr (str, ch)</code>	Searches str in for the last occurrence of ch. Returns a <i>pointer</i> to ch , or NULL if the character does not occur.

String Function List Continued

<code>char* strstr (haystack, needle)</code>	Searches haystack for needle. Returns a pointer to the position of needle within haystack, or NULL if the needle character string does not occur.
<code>char *strrstr (haystack, needle)</code>	Reverse search.
<code>char* strtok(str, delim)</code>	Returns a token string from str that is delimited by the string delim or NULL if not found. [1]
<code>int atoi (str1)</code>	Converts str1 to an integer and returns either the integer or 0 if str1 could not be converted to an integer.
<code>float atof (str1)</code>	Converts str1 to a floating point number and returns either the floating point number or 0 if str1 could not be converted to a float.
<code>itoa (value, str1, 10)</code>	Converts value to a base 10 string and puts it in str1. [2]

- // strings1.cpp
- // Simple program demonstrating the problems with overflow
- // Version
- // original by – Nicola Ritter
- // modified smr

- #include <cctype>
- #include <iostream>
- using namespace std;

- const int SIZE = 5;
- typedef char StringType[SIZE+1]; // why + 1

- //-----

Code

```
int main()
{
    stringType str1 = "abcde";
    stringType str2;
    str1 [ a b c d e \0 ]
```

```
cout << "str1: " << str1 << endl;
```

```
strcpy(str2, "0123456789");
```

```
cout << "str1: " << str1 << endl;
```

```
strcpy(str1, "vwxyz");
```

```
cout << "str1: " << str1 << endl;
```

```
cout << "str2: " << str2 << endl;
```

```
return 0;
```

Output

What output will we get?

str1: abcde

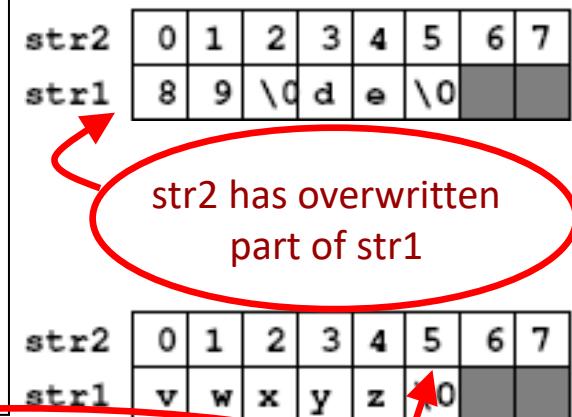
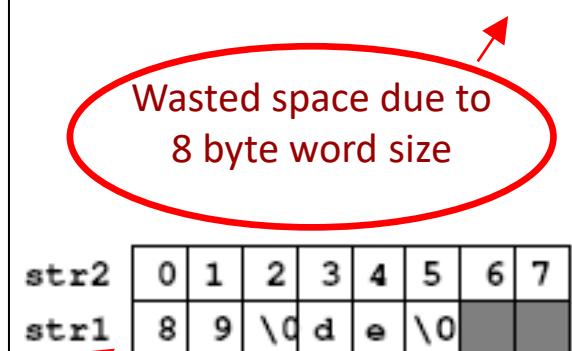
str1: 89

str1: vwxyz

str2: 01234567vwxyz

We have fixed up str1, but str2 still has no '\0' at the end of it.

Memory



Pointer Access to Strings

- // Output one character per line
- void CharOutput (const StringType str)
- {
- // Set a pointer at the beginning of the string
- // stop when it reaches the NULL character
- // the increment points the pointer at the next character
- for (char *ptr = str; *ptr != '\0'; ptr++) [1] //use nullptr
- {
- // Output the character at ptr
- cout << *ptr << endl;
- }
- }

Using Functions that Return Pointers

```
•  // Count the number of times str2 appears in str1

•  int Count (const StringType &str1, const StringType &str2)
•  {
•      int counter = 0;

•      // Prime the while loop by looking for a first occurrence
•      char *ptr = strstr(str1, str2);

•      // While we have found an occurrence of str2 in str1
•      while (ptr != NULL)
•      {
•          counter++;
•          ptr++; // go to the next char in str1. why? See next line
•          ptr = strstr (ptr, str2);
•      }

•      return counter;
•  }
```

Readings

- Textbook Chapter on Arrays and Strings, section on C-Strings
- Online reference to C string with examples
 - <http://en.cppreference.com/w/cpp/string/byte>
 - <https://www.cplusplus.com/reference/cstring/>
- Reminder:
 - It is essential to read the specified readings listed above. The lecture notes are not a substitute for the readings as the lecture notes do not cover everything you must know. See advice in unit guide.



Murdoch
UNIVERSITY

Data Structures and Abstractions

The STL, std string and more I/O

Lecture 9



What is the STL? [1]

- When C++ was written it was essentially a superset of C that allowed object oriented programming.
- At the start people either used the C-style arrays and strings (that we have already looked at and revised), or created their own classes.
- This was patently silly, so in the mid 1990s a standard set of templates was created for commonly used structures that are containers of data. [2]
- A container stores multiple objects of the same type. What changes between different types of container is the type of access, ordering and methods available.
- In this unit we will examine some of the most commonly used STL containers.

Problems with the STL

- The main problem with the STL is that it was clearly written by a committee.
- This means that:
 - It is ‘clunky’ – tries to please everyone.
 - Method names are often verbose, so everyone knows.
 - Use of methods may not intuitive all the time.
 - No bounds checking is done on arrays (vectors)!
 - The names of methods and classes is often non-mainstream. [1]
 - *Iterators* — which are generalisation of pointers — are used for a lot of the time, rather than indexes.
 - As it was written after C++ it might not always interface well with C++ and not all versions of C++ support it. [2]
 - But it is stable and usually well debugged – so can be used and usually very trusted.

The standard string class (C++)

- ANSI/ISO standard for C++ requires a string class to be a basic data type.
- Programmer should not have to worry about implementation details of the string class.
- It is part of the std namespace. [\[1\]](#)
- Is not part of C++ (as in reserved word); it is a programmer defined type available in the standard library.
- It has a wealth of good functions that make string handling much simpler.
- It has overloaded operators so that you can do such things as add to a string using the + operator.
- It makes reading in a whole line of text (including spaces) trivial.
- To use the standard string you must include the header file `<string>`.

The std string

- Unlike a C-style string, an std string cannot overflow.
- However no bounds checking is done on access.
- Do not assume that std string is NULL terminated. [1]
- Access to the string can be done as per any other string, using the [] operator.
- There are many good websites giving information on the various aspects of C++ including the std string.
- The following site is recommended.

<http://www.cppreference.com/cppstring/index.html>

String Operator List

In each of the examples below, str1, str3 and str4 must be std strings, however str2 can be a c-style string. // std:string str1, str3, str4 [1]

str1 = str2	Copies str2 to str1
str1 == str2 str1 < str2 str1 > str2	These give true or false depending on whether str1 is lexicographically equivalent to, less than or greater than str2.
str1 = str3 + str4	str1 becomes the concatenation of str3 and str4.
str1 += str2	str2 is appended to str1
str1 += ch	The character ch is appended to str1
cout << str1	str1 is output to screen
cin >> str1	The first word typed at the keyboard is stored in str1

Std::string methods

- Each method is overloaded multiple times.
- For example, to append to a string you could use:

<code>str1.append (str2)</code>	Appends str2 to the end of str1, where str2 can be a c-style OR std string.
<code>str1.append(str2, index, num)</code>	Append num characters from the part of str2 that starts at index.
<code>str1.append (str2, num)</code>	Append num characters from the start of str2, where str2 can be a c-style or std string.
<code>str1.append (num, ch)</code>	Add num repetitions of ch to the end of str1.
<code>str1.append (itr1, itr2)</code>	Append the string that starts at iterator itr1 and ends at iterator itr2.

List of Commonly Used Methods

- <various> below indicates that the method is overloaded and there are various possible parameters.
- For more information, go to
<http://www.cppreference.com/cppstring/index.html>
or your text book.

str1.append (<various>)	Append values str1.
str1.c_str ()	Returns a standard c-style string (char *). Required for file opening etc.
str1.clear ()	Empties the string.
str1.compare (<various>)	Compares two strings.
str1.copy (<various>)	Copies data into str1.
str1.empty ()	Returns true if the string is empty.
str1.erase (<various>)	Erases a part of the string.

Methods (continued)

<code>str1.find (<various>)</code>	Finds strings or characters within str1.
<code>str1.rfind (<various>)</code>	Finds strings/characters within str1, searching from the end.
<code>str1.insert (<various>)</code>	Inserts strings or characters into str1.
<code>str1.length ()</code> <code>str1.size ()</code>	Return the length of the string (number of objects in it)
<code>str1.push_back (ch)</code>	Add the character to the end of the string.
<code>str1.replace (<various>)</code>	Replace strings/characters within str1.
<code>str1.substr (index, num)</code>	Returns num characters starting from index.
<code>str1.swap (str2)</code>	Swaps the two strings.

The getline Function

- A really useful non-member string function.
- It allows the input of sentences—strings with white spaces—into a string:

```
string str;  
  
// Read in one word only  
cin >> str;  
  
// Read in characters until eoln  
getline (cin, str); [1]
```

- It is also possible to read in only part of a line, by specifying a delimiter. For example:

```
// Read in characters from keyboard until a comma  
getline (cin, str, ','); [2]  
// call the above as many times as needed to get each  
// comma separated piece of data. See notes below
```

Using Iterators

- All of the STL uses iterators for access to data. The same is true for std::string.
- For this reason, there are two functions that are standard for each container in the STL:

<code>.begin()</code>	Returns an iterator pointing at the first object in the container.
<code>.end()</code>	Returns an iterator pointing at the first memory location <i>past</i> the end of the container.

Simple string access program

Uses the
normal index
type access
method

```
• #include <string>
• #include <iostream>
• using namespace std;

• int main()
• {
•     string str;

•     cout << "Enter a string: ";
•     getline (cin, str);

•     cout << endl
•     << "The characters in your string were: "
•     << endl;
•     •
•     •
•     •

•     for (int index = 0; index < str.length(); index++)
•     {
•         cout << str[index] << endl;
•     }
•     cout << endl;
•     return 0;
• }
```

Simple string access program

Uses the iterator type access method

```
• #include <string>
• #include <iostream>
• using namespace std;

• int main()
• {
•     string str;

•     cout << "Enter a string: ";
•     getline (cin, str);

•     cout << endl
•     << "The characters in your string were: "
•     << endl;
• }

for (string::iterator itr = str.begin();
     itr != str.end(); itr++)
{
    cout << *itr << endl;
}
cout << endl;
return 0;
```

Testing

- Always test for:
 - empty strings;
 - empty files;
 - incorrect file names;
 - read-only files.
- Only assume data is correct if you have it stated in writing.
- Never assume the user won't make a mistake.
- Always have a test plan and run through it **every time** you alter the program in any way.

Text File I/O

- A name must be entered by the user.
- The file must be opened.
- A test must be done to check it has opened.
- Data is then read until EOF.
- File handling in C++ requires the `<fstream>` header file. See the lab exercises starting in topic 2.

- // text.cpp
- // Text file I/O
- // Reading a file of integers separated by spaces or new lines
- //
- // Version
- // 01 - Nicola Ritter
- // 02 – modified smr
- //-----
- #include <iostream>
- #include <fstream>
- using namespace std;
- //-----
- const int SIZE = 256;
- typedef char strType[SIZE+1]; // name the type
- //-----

- `int main ()`
- `{`
- `strType filename; // C array of chars`
- `// Get a file name from the user`
- `cout << "Enter name of file to read (one word only): ";`
- `cin >> filename;`
- `// Declare the file variable and try to open it using that`
- `// file name`
- `ifstream fstr (filename);`

```
•     // Is the ready state 0 (no errors)?
•     if (fstr.rdstate() == 0) // [1]
•     {
•         int number;
•
•         // prime the while loop
•         fstr >> number;
•         while (!fstr.eof())
•         {
•             cout << number << endl;
•             fstr >> number;
•         }
•
•     }
•     else
•     {
•         cerr << "Could not open file \""
•             << filename
•             << "\" for reading." << endl;
•     }
•
•     return 0;
• }
```

Opening Files using std::string

- `string filename;`
- `cout << "Enter filename: ";`
- `getline (cin, filename);`
- `ifstream fstr (filename.c_str());`
- `// The rest of the code is the`
- `// same as before`

File names can now contain spaces

`c_str()` gives the required access to the actual character string

`str.c_str()` can be also be used with some of the standard c-style string functions

More About File I/O

- Opening a file for output is done in a similar way, except that it will be an output file stream:
`ofstream fstr (filename);`
- To append to a file, you would open it with:
`ofstream fstr (filename, ios::app);`
- When you are going to store more of one type of data in a file, it is necessary to store the number of each type. You don't have to but this can cause problems in designing the algorithm. Algorithm would require asking the user or the value is hard coded.
- For example, when storing an array of 100 integers, you would write the 100 to the file first.
- When reading the file, you would then read the number 100 and therefore know that you had 100 integers to read.

Other Useful I/O Functions

- There are several generically useful I/O functions (as well as the formatting ones done previously) [1]:

<code>cin.good()</code>	Returns true if the stream (standard input or a file stream) has no errors.
<code>.flush()</code>	Flushes (empties) the stream (but not the actual file!).
<code>.peek()</code>	Peek at the next character without actually reading it.
<code>.ignore ()</code>	Ignore characters

Readings

- Chapter on Arrays and String, Section on C-Strings.
- Chapter on User-defined Simple Data Types, Namespaces, and the `string` type, section on `string` type.
- Chapter on Standard Template Library (STL)
- Website:
<http://www.cplusplus.com/faq/sequences/strings/split/>
- Website: “Iterators”,
<https://www.geeksforgeeks.org/introduction-iterators-c/>
- optional
- Video by Standford uni provided by academic earth. Links:
 - [C/C++ Libraries](#) explains string operations
 - [C++ Console I/O](#)



Murdoch
UNIVERSITY

Data Structures and Abstractions

Complexity

The STL Vector

[1] and

Iterators

Lecture 10



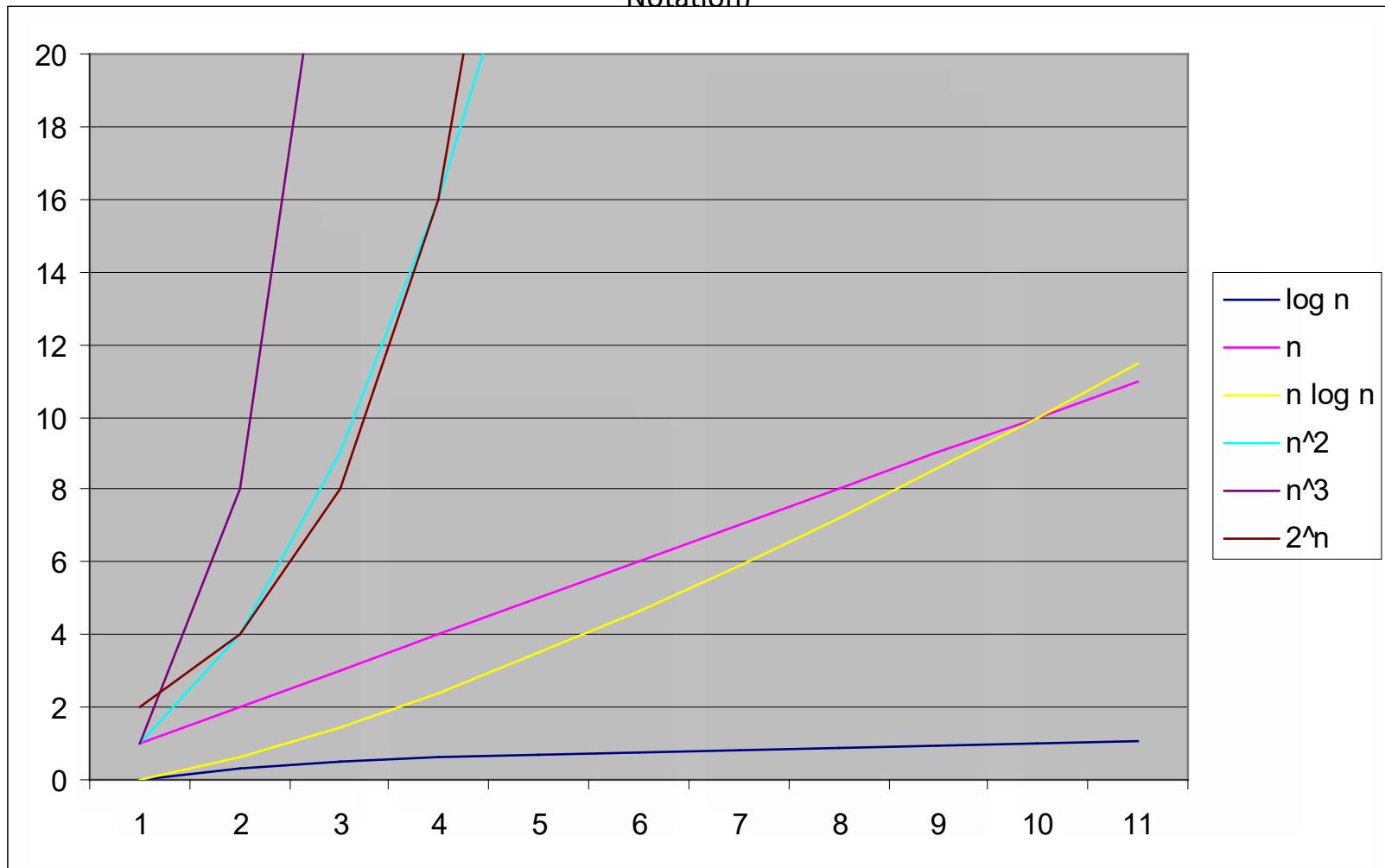
Complexity

- The complexity of algorithms is measured in terms of how long they take to execute or how much resources they utilise. Our current interest is on the execution time of the algorithm.
- This in turn is often measured in terms of the number of operations that are required.
- Complexity is described in two ways: descriptively and using O notation. [1]
- O notation means ‘in the order of’. It is that value multiplied by some constant. The order represents the rate of growth. The constant is something that can depend on a particular computer and can vary from computer to computer.
 - So we don’t consider this constant and only look at the algorithm.
- For all algorithms, the measurement is relative to how many items are being processed.
- The number of items is designated as ‘n’ and Big-O is a measure of the upper bounds of the number of operations carried out by the algorithm as n grows large.

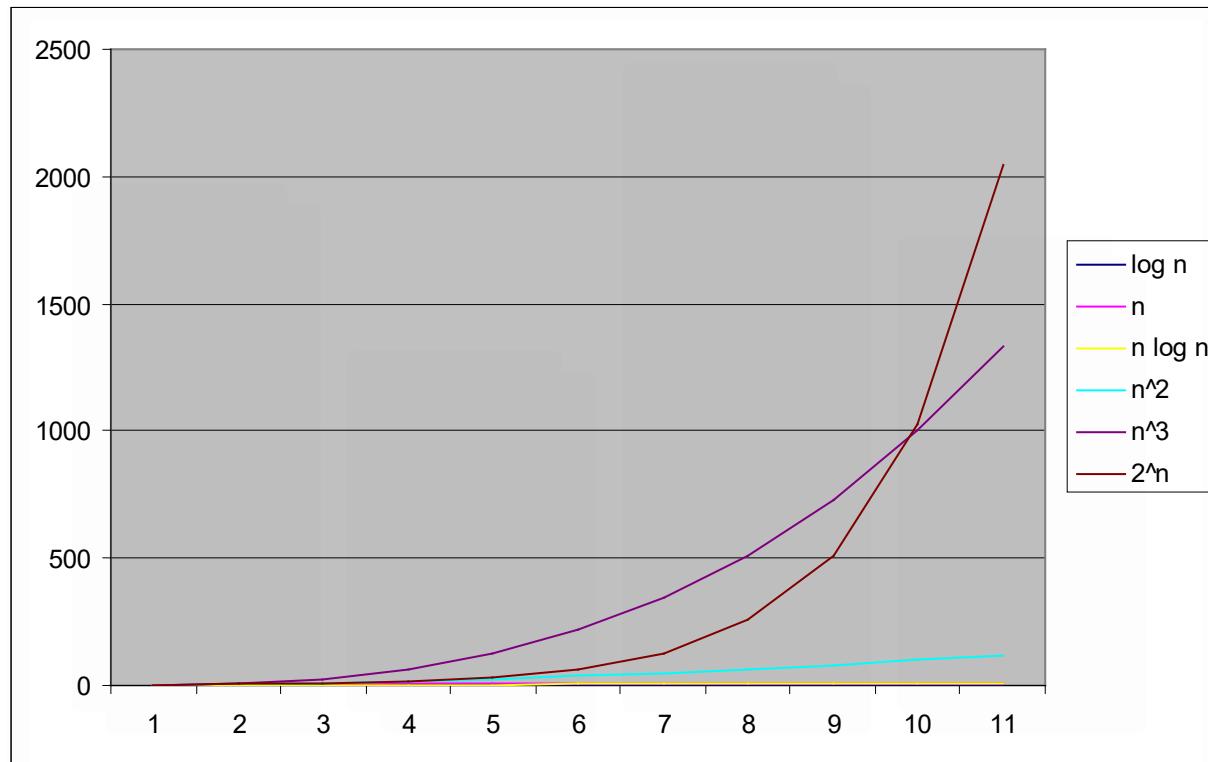
NAME	O notation	DESCRIPTION	Examples	n=1	n=10	n=20
constant time	k	It takes the same amount of time (k) no matter how many elements are being dealt with. If k is small this is the best case of all, as it will take the same time for 1 element as for 10,000.	$t = k$	k	k	k
logarithmic time	$O(\log n)$	The time taken increases very slowly as n increases.	$t = \log_{10}(n)$	0	1	1.3
linear time	$O(n)$	The time taken increases in a straight line as n increases.	$t = n$	1	10	20
	$O(n \log n)$	The time taken increases faster than n, but at a slower speed than the two below.	$t = n \log_{10}(n)$	0	10	26
polynomial time	$O(n^p)$	The time taken increases much faster than does n.	$t = n^2$ $t = n^3$	1 1	100 1000	400 8000
exponential time	$O(c^n)$	The time taken increases at a much, much higher rate than n. As n becomes very large, the time taken becomes almost infinite.	$t = 2^n$ $t = 10^n$	1 1	1024 10^{10}	1048576 10^{20}

Comparison at Scale 0-20

(textbook has a nicer diagram in the section on Big-O Notation)



Comparison at Scale 0-2500



The STL Vector [1] (can't be used for assignment 1 – not the same)

- The STL vector is an array-like template class, but not an array.
- It can be instantiated as any type that you choose, including ones designed by you.
- It has lots of inbuilt methods, as well as having functions within the algorithm class that operate on it.
- Note that, like all STL classes, there is *no* bounds checking done.
- To use the vector template, you must include the `<vector>` header file.
- Accessing a particular element or adding an element to a vector takes *constant* time.
- Finding an element or inserting an element at a particular location within the vector takes *linear* time.

Using the **STL** vector

- `#include <iostream>`
- `#include <iomanip>`
- `#include <vector>`
- `using namespace std;`
- `//-----`
- `// Declare a new type that is a vector (array) of integers`
- `typedef vector<int> IntVec;`
- `// Declare an iterator for this type of structure`
- `typedef IntVec::iterator IntVecitr;`
- `// Declare a new type that is a vector of vectors of integers`
- `// i.e. create a two dimensional array`
- `typedef vector<IntVec> IntTable;`
- `//-----`

- IntVec array;
- IntTable table;
- // Seeding a random number generator
- strand (time(NULL));
- // Adding random data to a vector
- for (int index1 = 0; index1 < SIZE; index1++)
- {
- // Add a number between 0 and 100 to the end of the array
- array.push_back (rand() % 100);
- }
- // Outputting the single array:
- for (int index3 = 0; index3 < SIZE; index3++)
- {
- cout << array[index3] << endl;
- }

```
•
•           // Make a table with identical rows
•           for (int index2 = 0; index2 < SIZE; index2++)
•           {
•               table.push_back (array);
•           }
•
•           // Outputting the table in columns
•           for (int row = 0; row < SIZE; row++)
•           {
•               for (int col = 0; col < SIZE; col++)
•               {
•                   cout << setw(5) << table[row][col] << " ";
•               }
•               cout << endl;
•           }
```

Vector Methods

- Like strings, vectors have many methods.
- Again like strings, most of the methods have multiple overloads.
- A good listing of information can be found at
<http://www.cppreference.com/cppvector/index.html>
- Unlike strings, there are only a few operators that apply to vectors.
- = and == are the two most useful operators that can be used with vectors.

- Only some examples shown – there are more.. [1]

vec.clear ()	Empties the vector.
vec.empty ()	Returns true if the vector is empty.
vec.erase (<various>)	Erases a part of the vector.
vec.insert (<various>)	Add data to the vector.
vec.push_back (data)	Add one piece of data to the end of the vector.
vec.pop_back ()	Delete the last item in the vector.
vec.begin()	Returns an iterator that points to the first item in the vector.
vec.end()	Returns an iterator that points to just after the last item in the vector.
vec.size()	Returns the size of the vector.
vec.swap (vec2)	Swaps the contents of the two vectors.

Vector Allocation

- When you `push_back` data into a vector, space needs to be allocated to the vector. This is done dynamically and the programmer doesn't need to worry about creating storage. This is normal for STL containers.
- The space is not allocated one 'slot' at a time.
- Instead it is allocated as follows: [1]

```
IF size > allocation/2
    allocate (size+1) new slots
ELSE
    allocate 1 new slot
```

- This results in the vector always being less than half full, which must give some efficiency advantage.
- However it means that vectors are space wasters.
- Furthermore, if the vector shrinks later, this does *not* result in released memory: the memory allocated stays allocated until the vector goes out of scope.

Example of Iterator Use

- Lets suppose we have a vector containing integers.
- We decide, for some reason, that we want to remove all the elements that are equal to some particular number, entered by the user.
- This will require:
 - An iteration through the vector
 - Erasure of each target as we find it
- The easiest way to do this is using an iterator.

```
• int target;  
• cout << "Enter number to be deleted: ";  
• cin >> target;  
•  
• IntVecItr itr = array.begin();  
• while (itr != array.end())  
• {  
•     if (*itr == target)  
•     {  
•         itr = array.erase(itr); [1] [2]  
•     }  
•     else  
•     {  
•         itr++;  
•     }  
• }
```

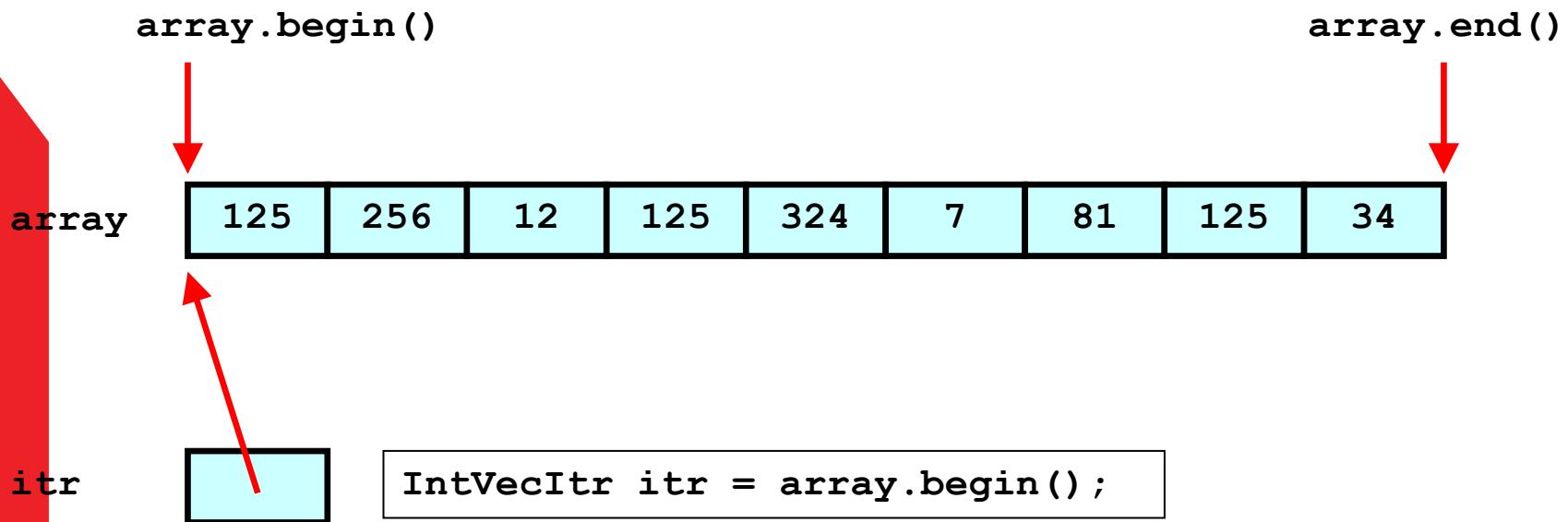
Defined earlier

erase
returns an
iterator to
the next
item

so we only need to
increment itr if we did
not delete an item

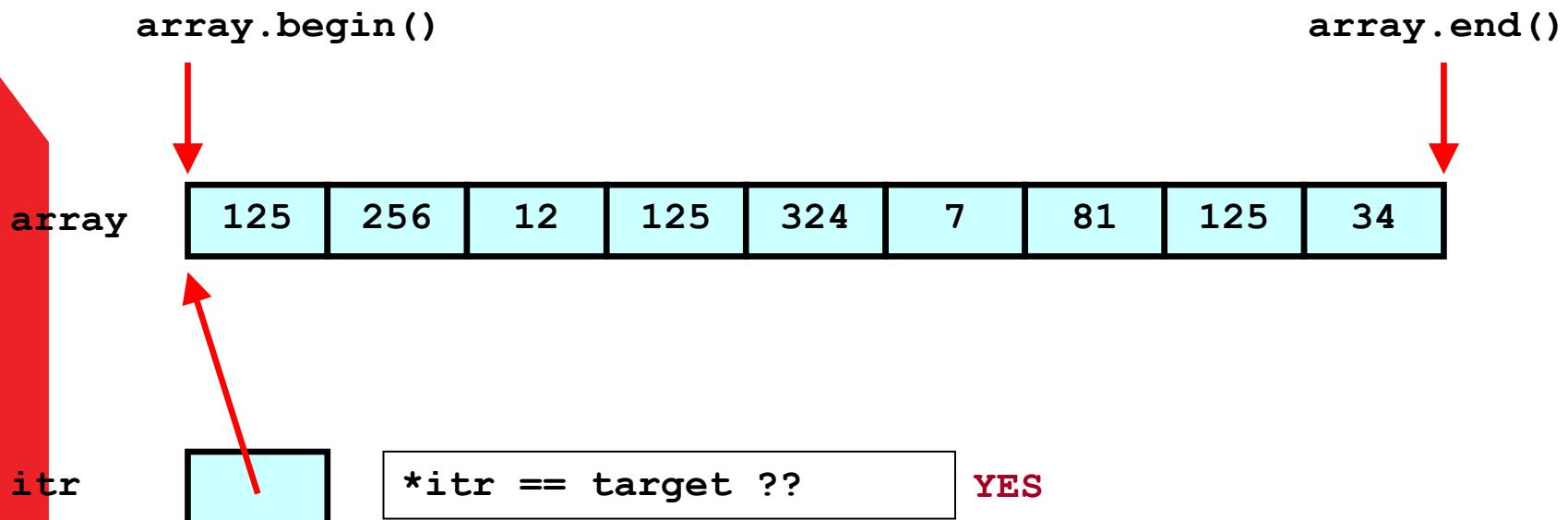
Example

target 125



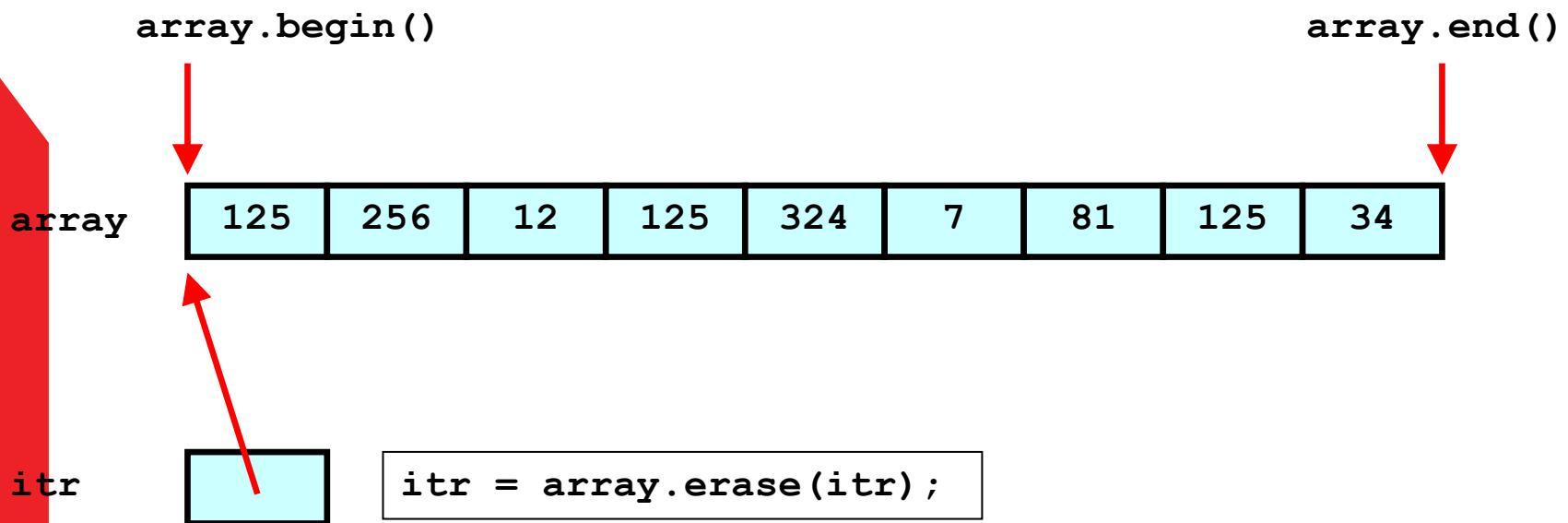
Example

target 125



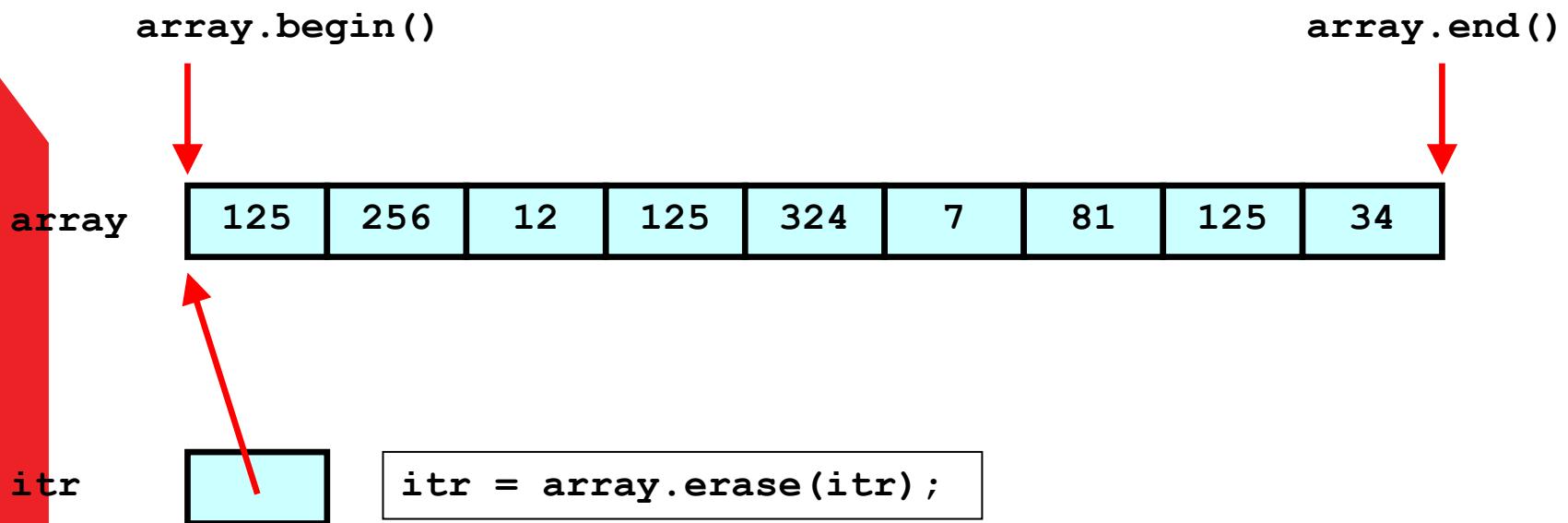
Example

target 125



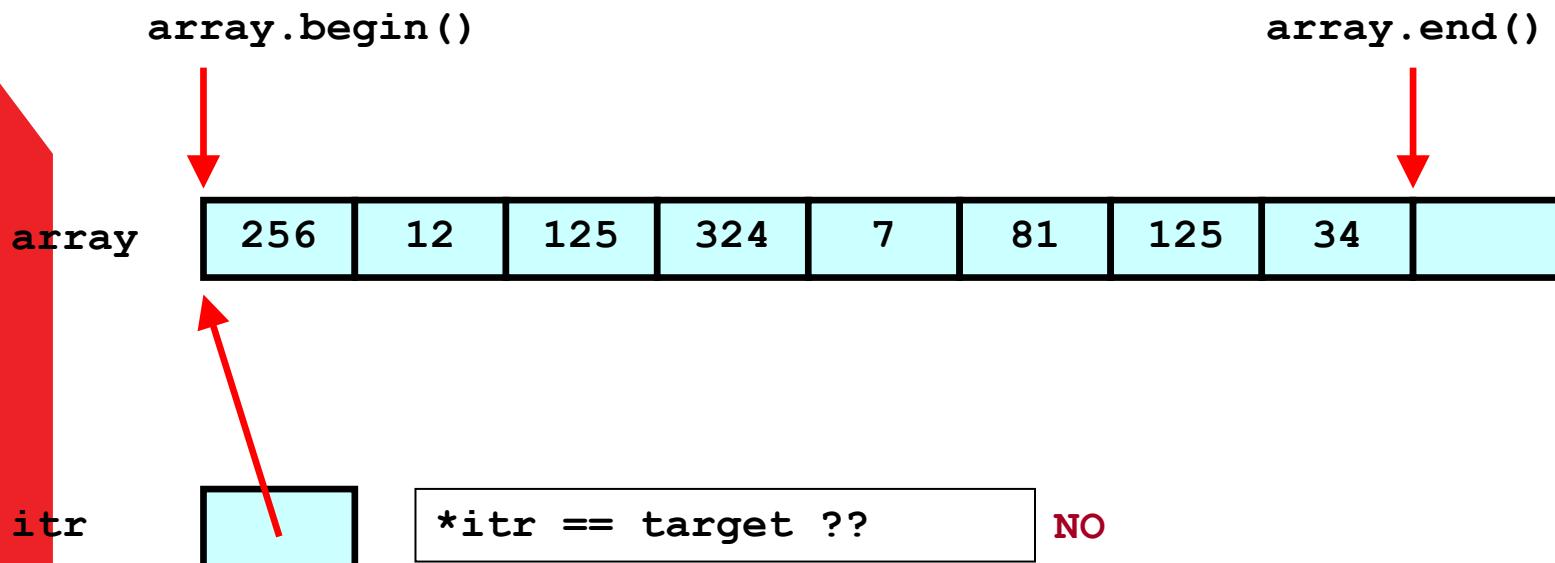
Example

target 125



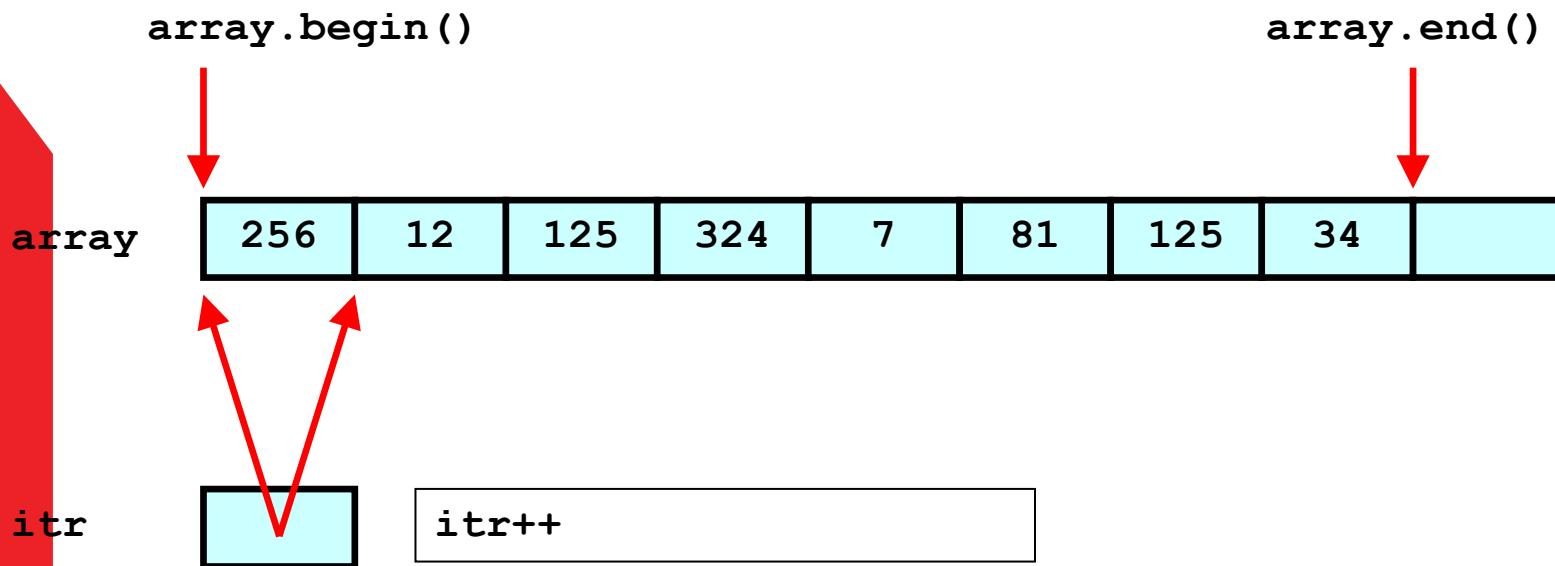
Example

target 125



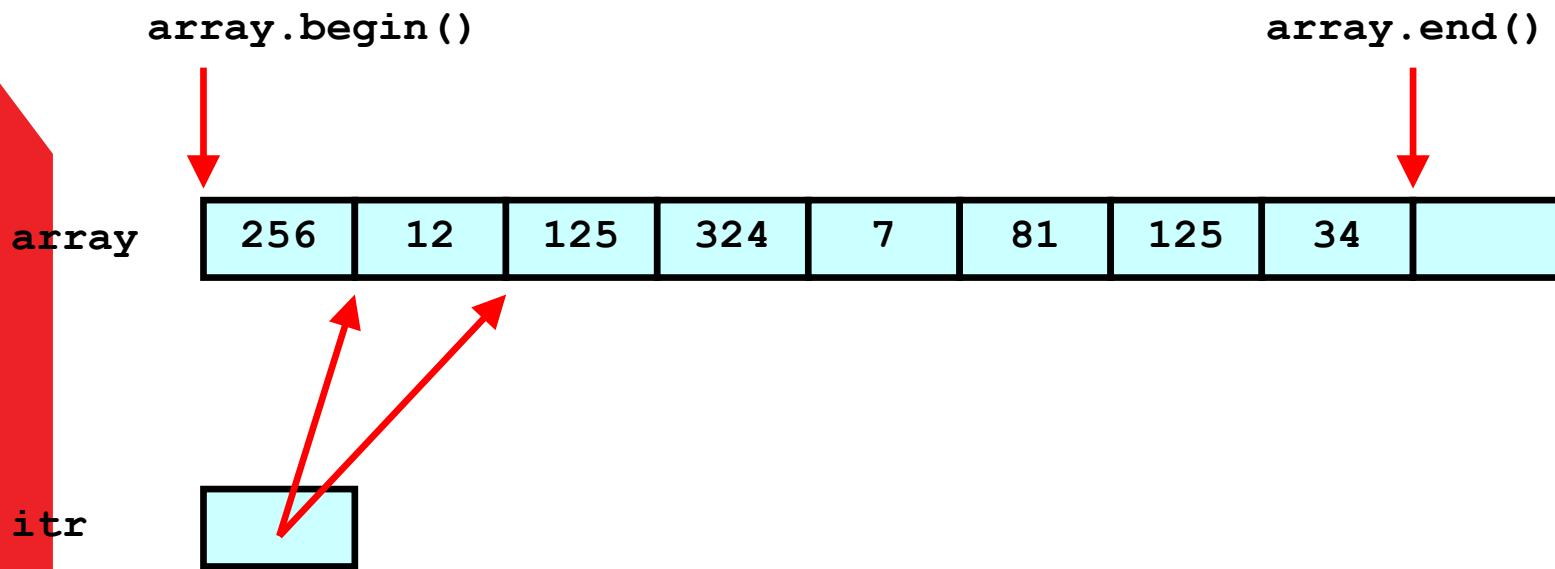
Example

target 125



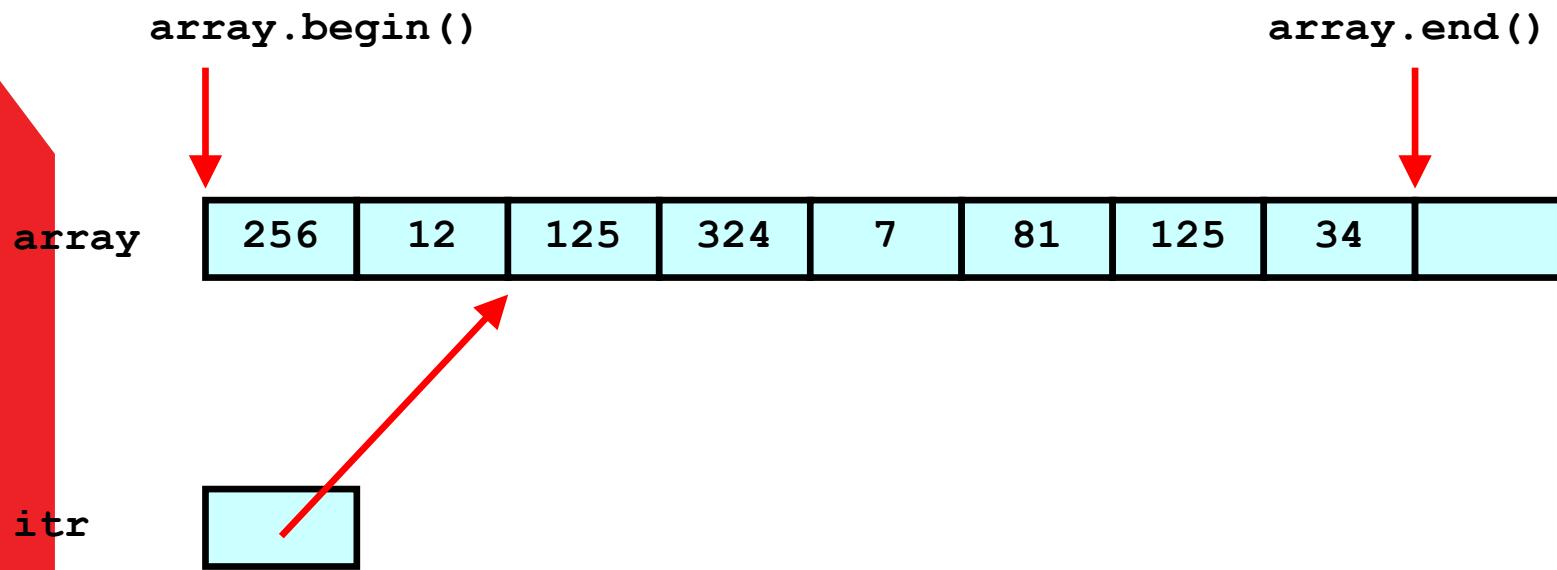
Example

target 125



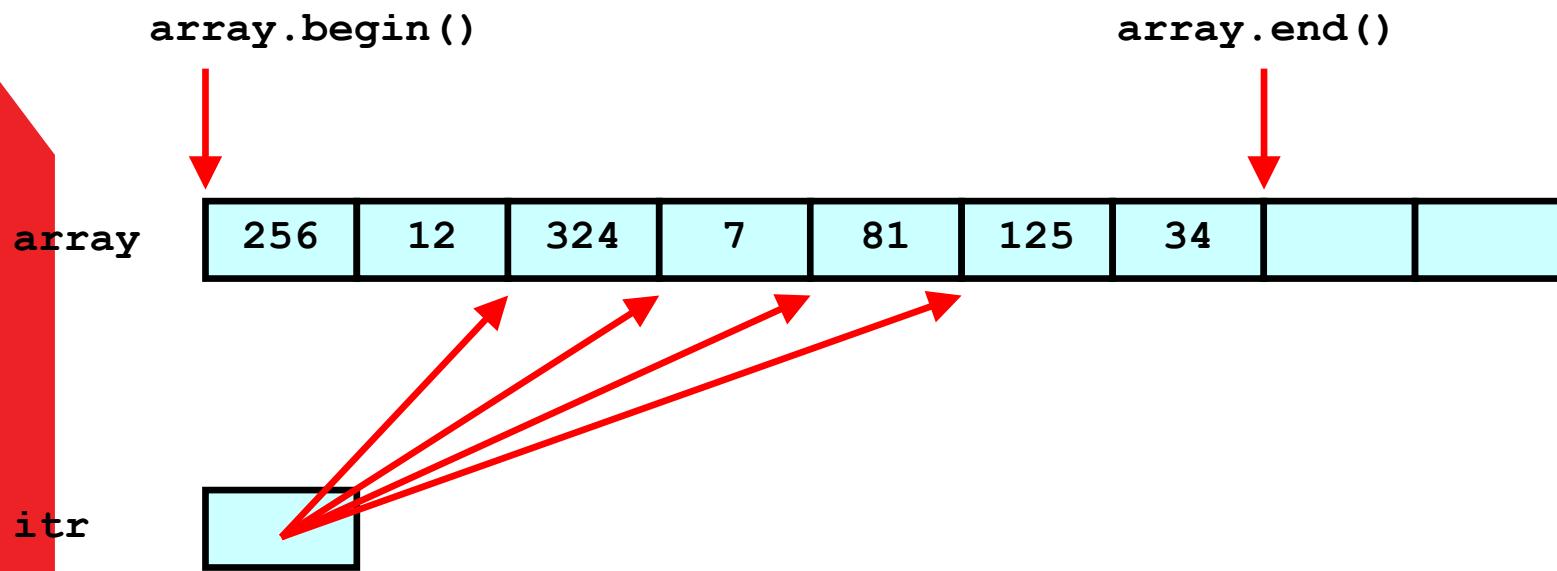
Example

target 125



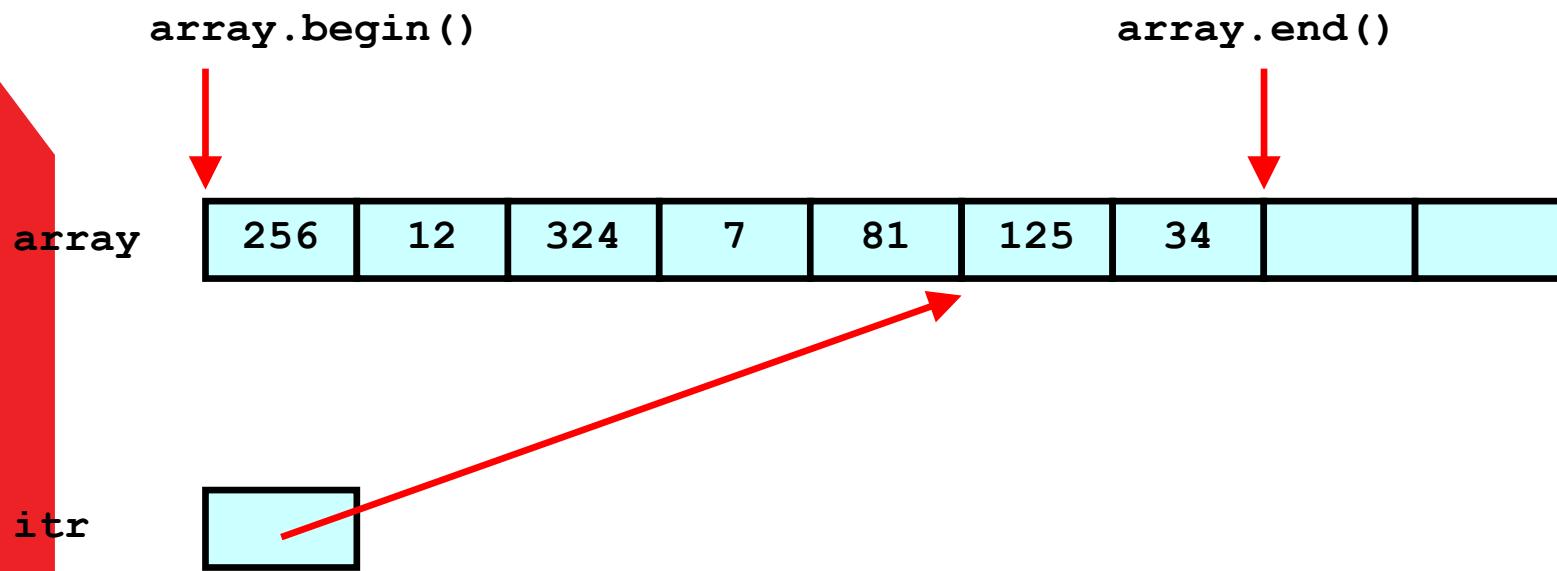
Example

target 125



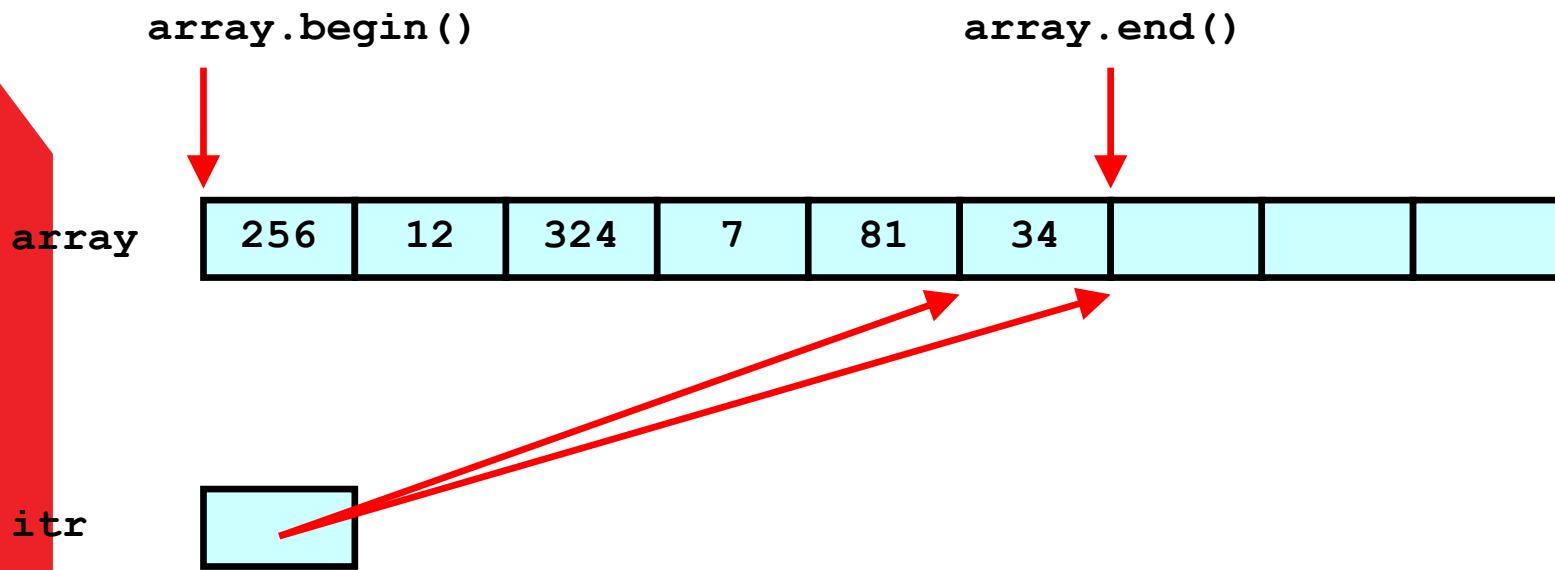
Example

target 125



Example

target 125



- What is the Big-O value for erase on vectors?

The Algorithm Class

- You may have noticed that there were no ‘find’ or ‘sort’ methods for the vector.
- That is because they are part of the STL algorithm class instead.
- To use this class, you must include the `<algorithm>` header file.
- The following code does the same repetitive delete as the previous code, but uses the ‘find’ algorithm.

- ```
int target;
```
- ```
cout << "Enter target to delete: ";
```
- ```
cin >> target;
```
- ```
IntVecItr itr = find (array.begin(), array.end(), target);
```
- ```
while (itr != array.end())
```
- ```
{
```
- ```
 array.erase(itr);
```
- ```
    itr = find (itr, array.end(), target);
```
- ```
}
```

- Note how much tighter (shorter) the code is now.

# Available Algorithms

- The algorithm class contains over forty algorithms.
- Please see the following links which have examples of use:  
<http://www.cppreference.com/cppalgorithm/index.html>  
<http://www.cplusplus.com/reference/algorithm/>
- The use of them assumes various operators are available for the data with which they are instantiated.
- This is no problem for a vector of integers or floats etc, as these already have all arithmetic and logical operators defined.
- Later on when we code classes, we will have to *overload* these operators if we wish to use the algorithm class's algorithms.

# Iterators Again

- If you actually have the index of something you want to delete or erase, you can ‘add’ index to the `.begin ()` iterator to get the correct thing to delete (or insert).

- For example:

```
array.erase (array.begin () +
10);
```

# The STL deque class

- Pronounced as “deck”
- The STL deque (double ended queue) class is almost identical to the vector class, except that it has as extra:
  - `push_front (const DataType &data)`
  - `pop_front ()`
- As well as the ones that work at the back.
- deque can grow dynamically at either end.
- Both of these functions work in constant time.

# deque

- Study the deque examples in the textbook in the chapter on “Standard Template Library”

# Exercise

## Using the STL vector as your data structure: [1]

- Write a simple program that will read in numbers from a file and output the mean and median to screen.
- Modify the program to output the standard deviation.

# Readings

- Textbook: Chapter on Searching and Sorting algorithms, section on Asymptotic Notation: Big-O Notation. If this is not found in your edition of the textbook please see the reference book “Introduction to Algorithms” chapter on Growth of Functions, section on O-notation.
- For a more comprehensive coverage of algorithms and their efficiency, see the reference book, Introduction to Algorithms. Chapters 1 to 3.
- Textbook Chapter on STL:
  - Sections related to the sequence container **vector** and **iterators** related to the sequence container vector.
  - The sequence container **deque** is also found in the above chapter.
  - **Skip** sections on ostream iterator and copy function.
- Website: “C++ Reference”, [\[1\]](http://www.cplusplus.com/reference/)
- Website: CPP reference, [\[2\]](http://www.cppreference.com/wiki/)



Murdoch  
UNIVERSITY

Data Structures and Abstractions

# Object Oriented Design and Testing

Lecture 11



# Object Oriented Terminology (Revision)

Note:

It is *object oriented*  
not object  
orientated!!!

- A *class*
  - is a description of a data type;
  - it includes (encapsulates) both data, and algorithms that operate on the data;
  - data should always be protected from being changed by anything other than one of the class's own algorithms;
  - the data members are also known as attributes or properties;
  - the algorithms are called methods rather than functions.
- An *object*
  - is a particular instance (example) of a class.

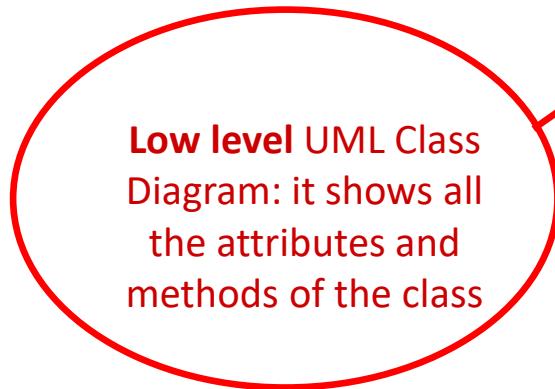
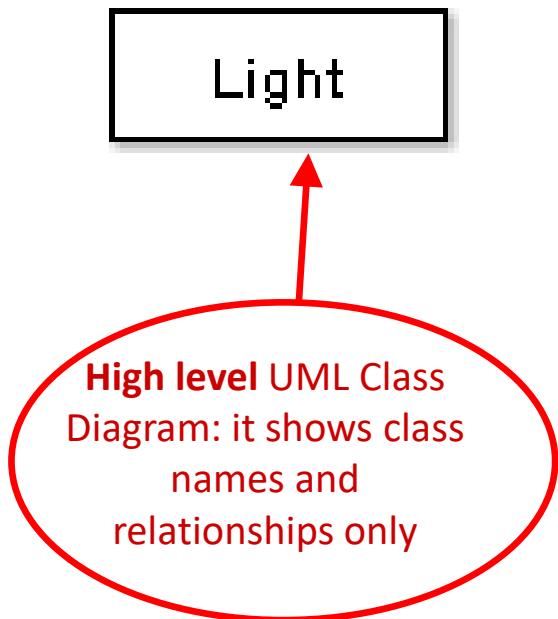
# Object Oriented Terminology (Revision)

- *Polymorphism, overloading, overriding*
  - In C++, refers to the use of the same name for more than one function or method;
  - In C++ polymorphism is used to specify abstract behaviour which may have different specific implementations. The version to invoke is determined at run-time. The abstract behaviour and specific behaviours are related by an inheritance hierarchy. This is just one example, typical for C++.
  - In C++ a child class can replace (override) a base class's method with its own version but this is not sufficient for polymorphism to occur.
  - C++ overloading is where the method or function name is the same but parameters can be different and the method or function to call is determined at compile time.
  - In C++ both methods and operators can be overloaded. Some operators cannot be overloaded. **Check the appendix of the textbook for more information.**

# Example Class

- Consider a class simulating a light.
- It might have attributes of:
  - `int colour`
  - `int radius`
  - `bool switchedOn`
- There might then need to be methods that (**not minimal**) [1]:
  - initialised all objects of the light class;
  - set each of the attributes;
  - returned the current value of each of the attributes;
  - output the current value of each attribute to the screen;
  - allowed input of values from the keyboard;
  - saved the current values to file;
  - read the current values from file;
  - etc. .. and the “kitchen sink”

# The Unified Modelling Language (UML) [1]



|                                                                                                                                                                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Light                                                                                                                                                                                                                                                                                     |
| <code>m_colour</code><br><code>m_radius</code><br><code>m_switchedOn</code>                                                                                                                                                                                                               |
| <code>Initialise()</code><br><code>SetColour()</code><br><code>SetRadius()</code><br><code>Switch()</code><br><code>GetColour()</code><br><code>GetRadius()</code><br><code>IsOn()</code><br><code>Input()</code><br><code>Output()</code><br><code>Read()</code><br><code>Write()</code> |

- While the low level diagram gives useful information, it results in *very* cluttered and hard to use diagrams. But if you are using a tool, then the low level diagram should be drawn and the more explicit descriptions can be given in a data dictionary. Don't forget to use -, + or #
- High level UML gives a good overview of your design.
- A data dictionary should also be provided.

# The Data Dictionary [1] [2] [3]

| Name                  | Type      | Protection | Description                                                                                    |
|-----------------------|-----------|------------|------------------------------------------------------------------------------------------------|
| Light                 |           |            | Simulates a light.                                                                             |
| m_colour              | integer   | +          | An integer light colour.                                                                       |
| m_radius              | float     | -          | The radius of the light.                                                                       |
| m_switchedOn          | boolean   | +          | True if the light is on.                                                                       |
| Initialise()          | procedure | #          | Sets the m_colour to white, m_radius to 0 and m_switchedOn to false.                           |
| SetColour(int colour) | boolean   | +          | If colour is positive it sets m_colour to colour and returns true. Otherwise it returns false. |
| SetRadius(float rad)  | Boolean   | Etc ...    | If radius is positive it sets m_radius to radius and returns true. Otherwise it returns false. |
| Switch()              | procedure |            | If the light is on, it switches it off. If the light is off it switches it on.                 |
| etc...                |           |            |                                                                                                |

# Object Oriented Relationships (revision) [1]

- There are 4 object oriented relationships that we use in this unit. We have encountered them before.
  - Association
  - Composition
  - Inheritance
  - Aggregation

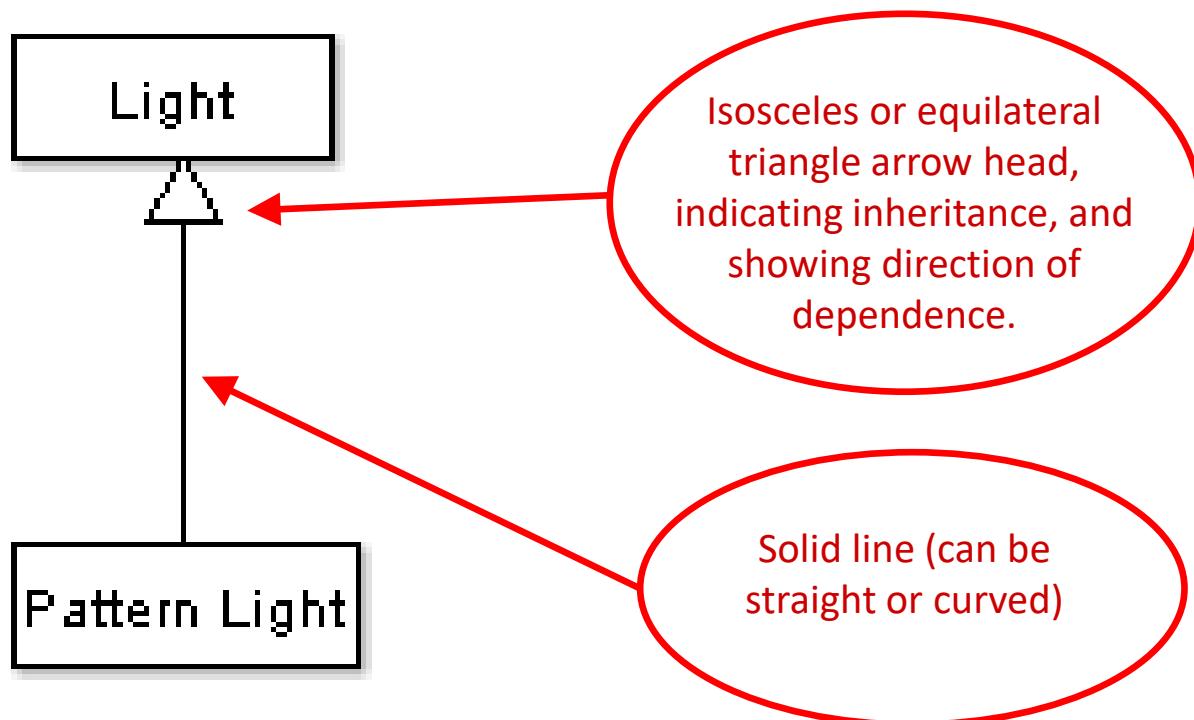
# Association

- Refers to the properties of a class. [1]
  - Normally, a basic type is depicted as an **attribute** and something more substantial is depicted as an **association** in an UML diagram. The meaning is still the same.
- An association is a very low-level generic term meaning that two classes are “somehow related”.
- At the lowest level this relationship can also mean that one class “uses” another class somewhere in its algorithms. **But see first point above.**
- Depicted with a **solid directed line**.

# Inheritance

- Any class may *specialise* or *extend* another class but just because the language lets you do this does not mean you should be doing it without considering the abstraction that you are trying to model. [1]
- We can say that one class “*is a*” variety of another class.
- We can *only* do this if it requires *all* of the data items *and* methods declared in the parent (base) class.

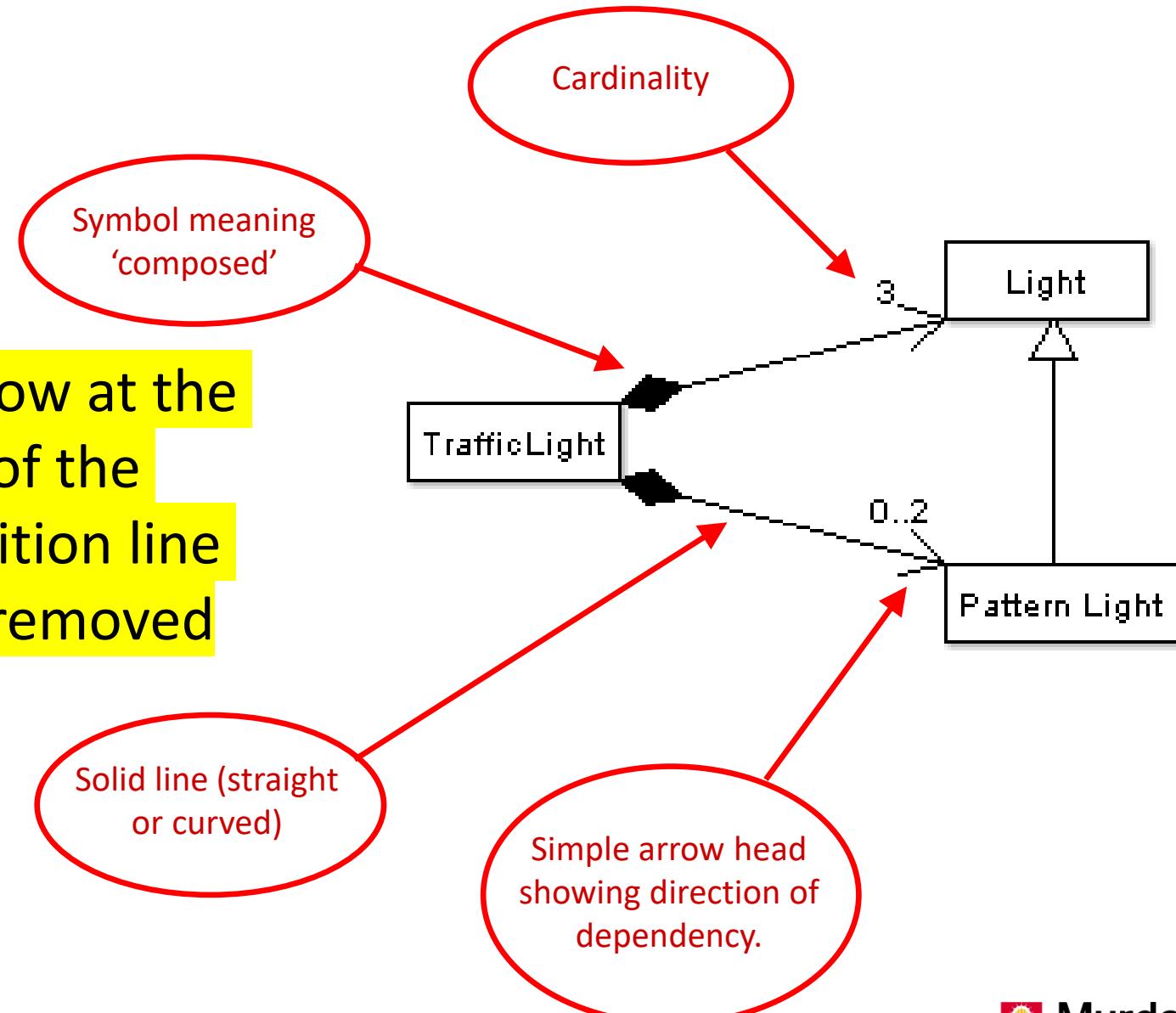
# Inheritance



# Composition

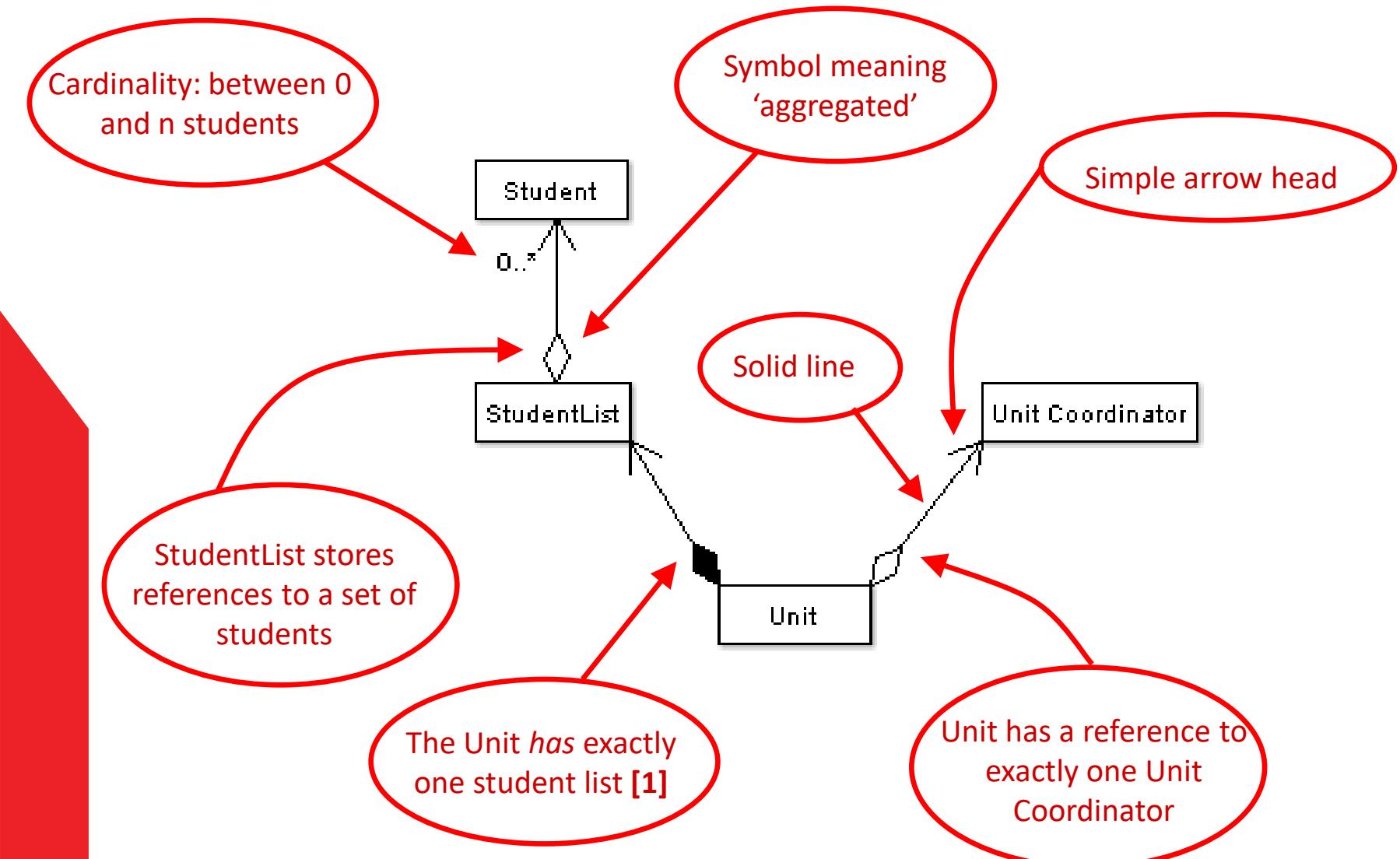
- Any class may be composed – in whole or part – of other classes.
- An instance must have only one owner object. There is no sharing with other owner objects.
- We can say that one class “*has a*” data item that is another class.
- For example, a class that emulates a simple traffic light would be composed of three lights, plus 0..2 lights that show arrows.

- The arrow at the end of the composition line can be removed



# Aggregation

- Some disagreement as to what this is. [1]
- We will use the following description.
  - Sometimes a class includes attributes that are pointers, references or keys to another class, but does not *control* the contained class.
  - For example, a Unit class might contain students and lecturers, but if the unit is deleted, the lecturers and students are not.
  - This is called aggregation.
  - In words, one could say that one class *refers* to another class, but is a kind of “contained” relationship.



# The Object Oriented Incremental Method

- What classes are required?
  - Consider all the data that is to be operated on by the program and try to split it into ‘things’ e.g. person, student, lecturer, unit, etc.
  - As you choose classes, place them in a UML diagram (use StarUML – it is free). **But you need to know how to draw these diagrams by hand as well.**
  - Identify containers (groups) of data, for example students, lecturers, etc.: these will form classes themselves.

- Order the classes from most depended upon to least depended upon.
- Then for each class (from the one most depended upon), *before* coding the next one in the list:
  - Code ***and test*** its
    - Initialiser (constructor)
    - Output method [1]
  - Code ***and test*** its
    - Set method(s)
    - Get method(s)
  - Code ***and test*** all other methods ***one by one.***

# Exercise

- A program is needed that will read in birth dates and output a person's age. What OO data structures might be required?
- If your Date class objects are stored in an array, how would sort the array, assuming you had a sort algorithm like bubble sort?
- A program is needed to read in time in UTC (Coordinated Universal Time). The time for output or display is to be for a given location (e.g. Perth or Sydney). How would you design the time class (or classes) for each location?

# Readings

- Textbook: Chapter on Classes and Data Abstractions. – **Please read now.**
- Reference book: *UML distilled: A Brief guide to the standard object modeling language* from My Unit Readings (chapters 1, 3 and 5) ebook from <https://murdoch.rl.talis.com/index.html>
- Rules for Design Style, Coding Style in unit Reference book, *C++ coding standards: 101 rules, guidelines, and best practices*. Also look at rules 32 to 44. ebook <https://go.exlibris.link/xwqTmlFk> (online)



Murdoch  
UNIVERSITY

Data Structures and Abstractions

# Object Orientation, Classes and Unit Testing (revision)

Lecture 12



# Object Orientation in C++

- In C++ we set up three files for each class:
  - The header file (.h) stores the class interface and data definition.
  - The implementation file (.cpp) stores the class implementation.
  - The (unit) test file (.cpp) tests every method for all parameter bounds.
- *Rules:*
  - Each class should represent only ONE thing. (cohesion)
  - Every class ***must*** be tested in isolation in a test file.
  - The testing ***must*** occur before the class is used by any other class. [1]
  - For every method in the class, you need to test all possible cases. This forms the class's Test Plan.

# The Light Class

- The previous lecture notes looked at a light class.
- This class stored information about a light: its colour, radius and whether it was switched on.
- We therefore start by coding a header file (`light.h`) with this information.
- Remember that we code and test *incrementally*.
- Therefore, we start the class with the bare minimum:
  - Constructor (initialiser).
  - Destructor (the opposite of the constructor). `//??` Needed [1]
  - Output operator to **test** that data is what we expect. `// for debugging only`
  - Attribute declarations.

- // Light.h
- // Class representing a light
- // Some methods shown, other methods you write.
- // Version
- // 01 - Nicola Ritter
- // modified smr
- //-----

```
#ifndef LIGHT_H }
#define LIGHT_H
//-----

#include <iostream>
#include <string> // OO string

using namespace std;
//-----
```

Ensures that this file is only included in the compilation once.

Class Name –  
capital first letter

```
• class Light ←
• { // only some methods shown. You write the other methods needed.
• // convert normal comments to doxygen style comments
• public:
• Light () {Clear();}
• ~Light () {};

• void Clear ();

• // provides a default output method, but be careful about friends
• // friends can be terrible, as they can mess up your privates
• friend ostream& operator << (ostream &ostr, const Light &light); // [1]

• private:
• // Any string giving a colour is acceptable
• string m_colour;
• // In centimetres
• float m_radius;
• bool m_on;
• };
• //-----
• #endif
```

- `class Light`
- `{`
- `public:` 
- `Light () {Clear ();}`
- `~Light () {};`
- `void Clear ();`
- `// friends are no good in most situations`
- `friend ostream& operator << (ostream &ostr, const Light &light);`
- `private:`
- `// Any string giving a colour is acceptable`
- `string m_colour;`
- `// In centimetres`
- `float m_radius;`
- `bool m_on;`
- `};`
- `//-----`
- `#endif`

public keyword:  
what follows is  
the interface.

```
• class Light
• {
• public:
• Light () {Clear();}
• ~Light () {};
•
• void Clear ();
• // can do without friends, ok for debugging purposes during development, but remove it
• friend ostream& operator << (ostream &ostr, const Light &light);
•
• private:
• // Any string giving a colour is acceptable
• string m_colour;
• // In centimetres
• float m_radius;
• bool m_on;
• };
•
• //-----
•
• #endif
```

private keyword:  
what follows is  
hidden from  
outside the class.

```
• class Light
• {
• public:
• Light () {Clear();}
• ~Light () {};
•
• void Clear ();
• // why have friends?
• friend ostream& operator << (ostream &ostr, const Light &light);
•
• private:
• // Any string giving a colour is acceptable
• string m_colour;
• // In centimetres
• float m_radius;
• bool m_on;
• };
•
• //-----
•
• #endif
```

all data must  
*always* be private  
or protected if  
you want sub-  
classes

- ```
class Light
{
public:
    Light () {Clear ();} [2]
    ~Light () {};
```
- ```
void Clear ();
// keep away from friends
friend ostream& operator << (ostream &ostr, const Light &light);
```
- ```
private:
    // Any string giving a colour is acceptable
    string m_colour;
    // In centimetres
    float m_radius;
    bool m_on;
};
```
- ```
//-----
```
- ```
#endif
```

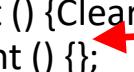
Constructor:
code is
automatically run
when an object is
declared. [1]

- ```
class Light
{
public:
 Light () {Clear ();} // [1] information hiding?
 ~Light () {};
```
- ```
void Clear ();
// friends can be overrated [2]
friend ostream& operator << (ostream &ostr, const Light &light);
```
- ```
private:
 // Any string giving a colour is acceptable
 string m_colour;
 // In centimetres
 float m_radius;
 bool m_on;
};
```
- ```
//-----
```
- ```
#endif
```

Inline code can  
be used if a  
method has  
exactly one line  
only. [1]

- class Light
- {
- public:
- Light () {Clear ();}
- ~Light () {};
- 
- void Clear ();
- //Even on facebook, friends can be no good
- friend ostream& operator << (ostream &ostr, const Light &light);
- 
- private:
- // Any string giving a colour is acceptable
- string m\_colour;
- // In centimetres
- float m\_radius;
- bool m\_on;
- };
- //-----
- #endif

Destructor: code is automatically run when an object goes out of scope.



```
• class Light
• {
• public:
• Light () {Clear();}
• ~Light () {};
•
• void Clear ();
• //Sun Tzu: Hold your friends close but your enemies closer
• //What does it say about having a "friend" so close that it is
• // inside the class.
• friend ostream& operator << (ostream &ostr, const Light &light);
•
• private:
• // Any string giving a colour is acceptable
• string m_colour;
• // In centimetres
• float m_radius;
• bool m_on;
• };
•
• //-----
•
• #endif
```

But nothing  
needs to be done  
to a Light object  
when it  
destructs, so the  
method is empty.  
[1]

```
• class Light
• {
• public:
• Light () {Clear();}
• ~Light () {};
•
• void Clear ();
• //Just because a language provides friendability, doesn't mean
• // friends can be used without proper justification. But can be used during debugging at development
• friend ostream& operator << (ostream &ostr, const Light &light);
•
• private:
• // Any string giving a colour is acceptable
• string m_colour;
• // In centimetres
• float m_radius;
• bool m_on;
• };
•
• //-----
• #endif
```

Every class needs  
a method that  
clears, resets,  
initialises or  
empties the  
object.

```
• class Light
• {
• public:
• Light () {Clear();}
• ~Light () {};
•
• void Clear ();
•
• friend ostream& operator << (ostream &ostr, const Light &light);
•
• private:
• // Any string giving a colour is acceptable
• string m_colour;
• // In centimetres
• float m_radius;
• bool m_on;
• };
•
• //-----
• #endif
```

The Clear() function is more than one line, so it is defined in the source file (follows).

- ```

class Light
{
public:
    Light () {Clear();}
    ~Light () {};

    void Clear ();
    // No friends – enough said
    friend ostream& operator << (ostream &ostr, const Light &light); [1]

private:
    // Any string giving a colour is acceptable
    string m_colour;
    // In centimetres
    float m_radius;
    bool m_on;
};

//-----
```
- #endif

‘friend’ operators and methods are those that link two different classes, in this case ostream and Light

```
• class Light
• {
•     public:
•         Light () {Clear();}
•         ~Light () {};
•
•         void Clear ();
•         friend ostream& operator << (ostream &ostr, const Light &light);
•
•     private:
•         // Any string giving a colour is acceptable
•         string m_colour;
•         // In centimetres
•         float m_radius;
•         bool m_on;
•     };
•
•     //-----
•
• #endif
```

The return value
of this operator is
a reference to an
output stream.



```
• class Light
• {
•     public:
•         Light () {Clear();}
•         ~Light () {};
•
•         void Clear ();
•         friend ostream& operator << (ostream &ostr, const Light &light);
•
•     private:
•         // Any string giving a colour is acceptable
•         string m_colour;
•         // In centimetres
•         float m_radius;
•         bool m_on;
•     };
•
• //-----
• #endif
```

The operator we are *overloading* is the standard output operator. [1]



```
• class Light
• {
•     public:
•         Light () {Clear();}
•         ~Light () {};
•
•         void Clear ();
•         friend ostream& operator << (ostream &ostr, const Light &light);
•
•     private:
•         // Any string giving a colour is acceptable
•         string m_colour;
•         // In centimetres
•         float m_radius;
•         bool m_on; // [1] don't start with _ in user classes
•     };
•
•     //-----
• #endif
```

All attributes are
prefaced with m_
for 'member'

```
• class Light
• {
•     public:
•         Light () {Clear();}
•         ~Light () {};
•
•         void Clear ();
•         friend ostream& operator << (ostream &ostr, const Light &light);
•
•     private:
•         // Any string giving a colour is acceptable
•         string m_colour;
•         // In centimetres
•         float m_radius;
•         bool m_on;
•     };
•
•     //-
• #endif
```

Don't miss out
the semicolon [1]

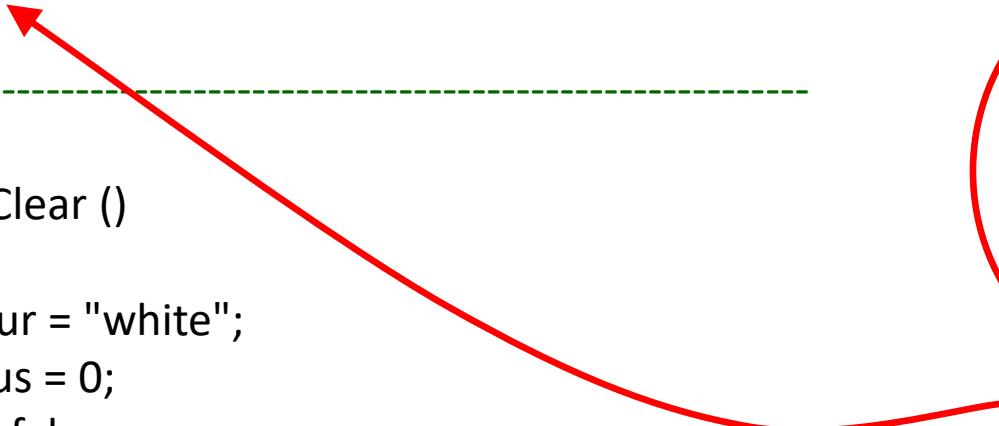
- ```
class Light
{
public:
 Light () {Clear();}
 ~Light () {};
```
- ```
void Clear ();
friend ostream& operator << (ostream &ostr, const Light &light);
```
- ```
private:
 // Any string giving a colour is acceptable
 string m_colour;
 // In centimetres
 float m_radius;
 bool m_on;
};
```
- ```
//-----
```
- ```
#endif
```

Matches the  
#ifndef on  
previous slide

# Light Definition

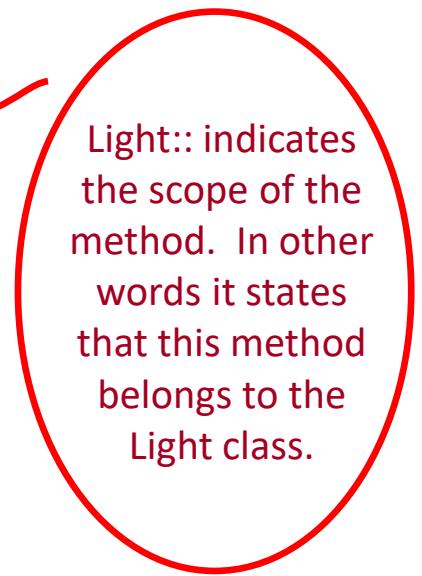
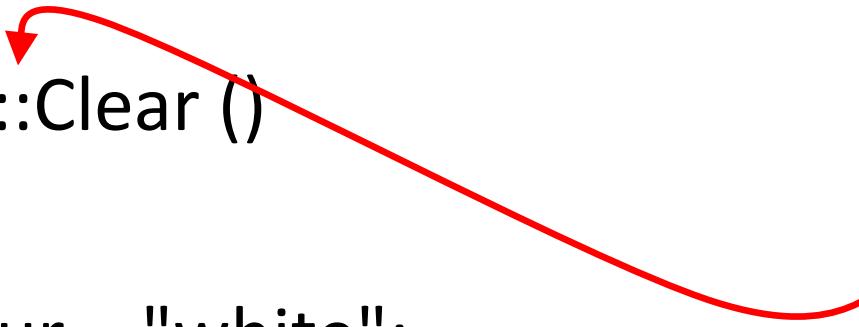
- There was one method (**Clear()** ), and one operator (**<<**) that were more than one line long and hence were not be defined in the header file. It is best not to have implementation in the class declaration. Separate the interface (declaration) from the implementation [1]
- Such methods and operators are defined in a source file (.cpp) with the same base name as the header (.h) file.

- // Light.cpp – implementation is separated from the interface/specification .h file
- //-----
- #include "Light.h"
- //-----
- void Light::Clear ()
- {
- m\_colour = "white";
- m\_radius = 0;
- m\_on = false;
- }
- //-----



The header file is included as a local rather than system header file. [1]

- *// Light.cpp*
- **void Light::Clear ()**
- {
- **m\_colour = "white";**
- **m\_radius = 0;**
- **m\_on = false;**
- }
- *//-----*



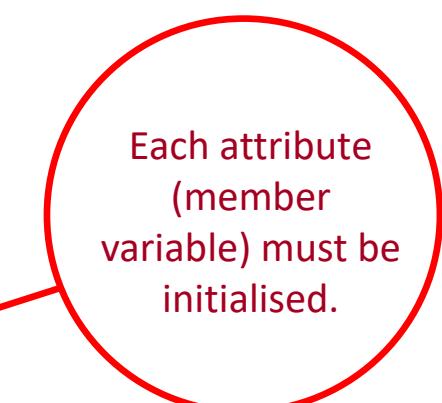
Light:: indicates the scope of the method. In other words it states that this method belongs to the Light class.

- `// Light.cpp`

- `//-----`

- `void Light::Clear ()`
- `{`
- `m_colour = "white";`
- `m_radius = 0;`
- `m_on = false;`
- `}`

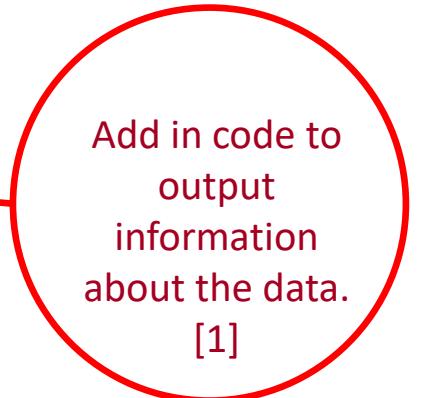
- `//-----`



Each attribute  
(member  
variable) must be  
initialised.

- ```
ostream &operator << (ostream &ostr, const Light &light)
{
    ostr << light.m_radius << " cm "
    << light.m_colour << " light is ";
    if (light.m_on)
    {
        ostr << "on";
    }
    else
    {
        ostr << "off";
    }

    return ostr;
}
```
- //-----



Add in code to output information about the data.
[1]

Testing

- *“Somehow at the Moore School (UPenn) and afterwards, one had always assumed there would be no particular difficulty in getting programs right. I can remember the exact instant in time at which it dawned on me that a great part of my future life would be spent finding mistakes in my own programs.”* – Maurice Wilkes, Computer Science pioneer and inventor of microprogramming.
- A lot has changed since Wilkes' keen observation as the amount of time and resources spent on testing has increased considerably; there now IT job roles called “Testers” and tools for automated testing.
- **You should not incorporate any subroutine, method, class or software component into the main build for the application unless incoming software is fully tested. There is a heavy price to be paid for ignoring this advice.**

The Test File

- So far we have a header file and a source file which together completely define a class.
- However, we do not have a program because we have no `main()` function. [1]
- Nor do we have a test plan and this is very bad!! Test plans should exist when you have understood the specifications and before coding. After and while coding, you may want to add to the existing test plan.

Initial Test Plan

Test	Description (including why the test is needed)	Actual Test Data	Expected Output	Passed
1	Check that constructor initialises the data and check that output operator works.	NA – default constructor	0 cm, white light is off	
2	Check that setRadius works	2.5 is sent as parameter	<i>...fill in what is expected</i>	

Complete the class methods and then we can write a program that uses the class and follows the test plan. Do the testplan in a table or in a spreadsheet.

- // LightTest.cpp This is the unit test program. To compile it needs only the
- // .h file. To link, it will need the .cpp [1]
- //-----

- #include "Light.h"

- //-----

- int main()
- {

- Light light;

- cout << "Light Test Program" << endl << endl;
- cout << "Test One" << endl;
- cout << light; // [2]

- cout << endl;
- return 0;

- }

The destructor runs when the block in which the light was declared, ends.

Runs the code in the constructor, i.e. the Clear () function.

Runs the code in the friend output operator

LightTest.cpp

- LightTest.cpp is the test program for the light class. [1]
- When it runs it will take the tester through all the tests in the test plan.
- For each test there must be output letting the tester know which test is being run.
- When it is run, the appropriate ticks are placed in the test plan's 'passed' column.
- It is refactored regularly as the test plan grows.
- It is run in its entirety every time anything is changed in the class.
- Which means that you need to print a new copy of the test plan each time you make a change.
- It is your proof that the class you have written is without bugs.
- It allows you to say with confidence "this class is finished".

Readings [1].

- Go through the following In Absolute C++. Pages 284-293 very carefully. Via My Unit Readings (log in) . If the link is not working, contact the Murdoch University Library.
- Textbook, Chapter on Classes and Data Abstraction
- My Unit Readings: Testing and debugging (Chpt. 8). [View Online at library site](#). You would struggle to finish the data structure unit/module if you can't write test harness and do unit tests for the software that you write.
- Chapter on Pointers, Classes, Virtual Functions, Abstract Classes and Lists, section on Shallow versus Deep Copy and Pointers; section on Classes and Pointers: Some Peculiarities.
- Chapter on Overloading and Templates, all the sections till (and including) section on Function Overloading.
- Rules for Design Style, Coding Style in unit Reference book, [C++ coding standards: 101 rules, guidelines, and best practices](#). Also look at rules 32 to 44. Library ebook <https://go.exlibris.link/xwqTmlFk>



Murdoch
UNIVERSITY

Data Structures and Abstractions

Completing the Minimal and Complete Class

Lecture 13



Versioning and Backing Up

- Of course, as you add to the class you must:
 - **Backup the previous version before changing anything.**
 - Change the version number and describe what you are changing.
 - Test *everything* in the test plan again.
- The easiest way to store the backups is simply to have a directory called ‘backup’ and label the backups (presumably zip files) with the version number.
For example: Light-01.zip, Light-02.zip etc.
- The zip file will contain the *entire* workspace, making reversion to a backup simple.

Versioning and Backing Up

- The previous backup and versioning method is **manually intensive**.
- There are automated tools available which help with this.
- You are encouraged to try out a tool like git <http://git-scm.com/>, Subversion <http://subversion.tigris.org/> or Redmine <http://www.redmine.org/>. You might want to try a demo of Redmine at <http://demo.redmine.org/>.
- Subversion is server software and you as the user connect to it using a client which runs on your machine. One example is TortoiseSVN <http://tortoisessvn.tigris.org/>.
- **We recommend the following if you can't make up your mind.**
 - GitHub <https://education.github.com/> or
 - Bitbucket <https://education.github.com/>
- Make sure that the repositories are private.

Code & Test Plan

The additions are made in order:

- Set methods for each data member.
- Get methods for each data member.
- Overloaded assignment operator.
- Copy constructor.
- If **required** add:
 - overloaded input operator;
 - overloaded relational operators;
 - overloaded arithmetic operators;
 - file I/O methods;
 - other overloaded constructors;
 - processing methods

This list may not be the minimal set.

Should these be part of the class?
Think through this carefully.
I/O operators would not normally be part of the class – see previous topics. So where would you implement them? [1]

Change Plan

For each change or addition to a class, you must:

- Add the method descriptions to the header (.h) file.
- Add the code to the implementation (.cpp) file.
- Add tests to the test plan.
- Add tests to the test program (**unit test** program).
- Run **all** the tests every time and debug the code

Changes to the Header File

- // Light.h
- // Class representing a light
- //
- // Version
- // 01 - Nicola Ritter date ..
- // 02 – Nicola Ritter, date ...
- // Added in Set methods
- // 03 – smr, date..
- // to convert all friend methods to non-friends,
- // non-members
- //-----

It is very important to record what you changed!

Use Doxygen style comments in header (.h) files



- `// Needs doxygen comments`
- `// separate the implementation – remove from inside the class`
- `class Light`
- `{`
- `public:`
- `Light () {Clear ();}`
- `~Light () {};` **//[1]**
- `void Clear ();`
- `void SetColour (const string &colour) {m_colour = colour;}`
- `bool SetRadius (float radius);`
- `void Switch () {m_on = !m_on;}`
- `friend ostream& operator << (ostream &ostr, const Light &light);` **[2]**
- `private:`
- `// Any string giving a colour is acceptable`
- `string m_colour;`
- `// In centimetres`
- `float m_radius;`
- `bool m_on;`
- `};`

We are accepting any string as a colour, so there is no error state

Inline function, switches the light to the opposite of what it was

Changes to the Implementation (.cpp) File

- `//-----`
- `bool Light::SetRadius(float radius)`
- `{`
- `if (radius > 0)`
- `{`
- `m_radius = radius;`
- `return true;`
- `}`
- `else`
- `{`
- `return false;`
- `}`
- `}`
- `//-----`

Changes to the Test Plan [1]

Test	Description	Actual test call/data	Expected Output	Passed
1	Check that constructor initialises the data and check that output operator works.	Light light	0 cm, white light is off	
2	Colour setting works.	light.SetColour ("red")	0 cm, red light is off	
3	Setting a negative radius will fail.	light.SetRadius (-9.3)	Error message 0 cm, red light is off	
4	Setting with a positive radius will work.	light.SetRadius (9.3)	9.3 cm, red light is off	
5	Switching an off light on.	light.Switch ()	9.3 cm, red light is on	
6	Switching an on light off.	light.Switch ()	9.3 cm, red light is off	
7	Clearing a light that is on.	light.Switch() light.Clear ()	0cm, white light is off	

Changes to the Test File (Unit Test)

```
• #include "Light.h"
• using namespace std; // expose everything – not good but it is convenient for now.
•
• int main()
{ 
    Light light;
    cout << "Light Test Program" << endl << endl;

    cout << "Test One" << endl;
    cout << light << endl << endl;

    cout << "Test Two" << endl;
    light.SetColour("red");
    cout << light << endl << endl;

    cout << "Test Three" << endl;
    if (!light.SetRadius((float)-9.3))
    {
        cerr << "Radius must be greater than 0" << endl;
    }
    cout << light << endl << endl;
```

The (float) is called a 'cast'. Without it, the compiler assumes 9.3 is a double rather than a float, and a warning is generated stating "truncation from 'const double' to 'float'" [1]



```
cout << "Test Four" << endl;
if (!light.SetRadius((float)9.3))
{
    cerr << "Radius must be greater than 0" << endl;
}
cout << light << endl << endl;

cout << "Test Five" << endl;
light.Switch();
cout << light << endl << endl;

cout << "Test Six" << endl;
light.Switch();
cout << light << endl << endl;

cout << "Test Seven" << endl;
light.Switch();
light.Clear();
cout << light << endl << endl;

cout << endl;

return 0;
}
```

This is about as long as a function should get.

If more tests get added, then `main()` must become a function that calls other functions that do the actual tests.

Each test can be in its own function. Makes things a lot neater.

Output From LightTest

- Light Test Program
- Test One
- 0 cm, white light is off
- Test Two
- 0 cm, red light is off
- Test Three
- Radius must be greater than 0
- 0 cm, red light is off

Test Four

9.3 cm, red light is off

Test Five

9.3 cm, red light is on

Test Six

9.3 cm, red light is off

Test Seven

0 cm, white light is off

Refactored LightTest.cpp

```
// LightTest.cpp
// modularised unit test - preferred way so that each
// test number matches the test plan number.
// Approach for unit testing classes for assignment

//-----
#include "Light.h"

//-----
void Test1 () ; //print after construction
void Test2 () ; // set colour
void Test3 () ;
void Test4 () ;
void Test5 () ;
void Test6 () ;
void Test7 () ;
```

It is also a good idea to comment each call to indicate what the test is doing. Get this comment from the test plan table.

- `//-----`
- `int main()`
- `{`
- `Light light;`
- `cout << "Light Test Program" << endl << endl;`
- `Test1(); //print after construction`
- `Test2(); // set colour`
- `Test3();`
- `Test4();`
- `Test5();`
- `Test6();`
- `Test7();`
- `cout << endl;`
- `return 0;`
- `}`

It is also a good idea to comment each call to indicate what the test is doing.
Copy/Paste from testplan table

- //-----
- `void Test1 () // comment each test here too - copy/paste from testplan table`
- {
- `Light light;`
- `cout << "Test 1" << endl; // make it more descriptive`
- `cout << light << endl << endl;`
- }
- //-----
- `void Test2 () // set colour`
- {
- `Light light;`
- `cout << "Test 2" << endl;`
- `light.Set("red");`
- `cout << light << endl << endl;`
- }
- etc...

Get Methods [1]

- `public:`
- `Light () {Clear ();}`
- `~Light () {}`
- `void Clear ();`
- `void SetColour (const string &colour) {m_colour = colour;}`
- `bool SetRadius (float radius);`
- `void Switch ();`
- `void GetColour (string &colour) const {colour = m_colour;}`
- `float GetRadius () const {return m_radius;}`
- `bool IsOn () const {return m_on;}`
- etc.

Get Methods

```
public:  
    Light () {Clear ();}  
    ~Light () {};  
  
    void Clear ();  
  
    void SetColour (const string &colour) {m_colour = colour;}  
    bool SetRadius (float radius);  
    void Switch ();  
  
    void GetColour (string &colour) const {colour = m_colour;}  
    float GetRadius () const {return m_radius;}  
    bool IsOn () const {return m_on;}  
  
etc.
```

Objects are returned parameter-wise.

Get Methods

```
public:  
    Light () {Clear ();}  
    ~Light () {};  
  
    void Clear ();  
  
    void SetColour (const string &colour) {m_colour = colour;}  
    bool SetRadius (float radius);  
    void Switch ();  
  
    void GetColour (string &colour) const {colour = m_colour;}  
    float GetRadius () const {return m_radius;}  
    bool IsOn () const {return m_on;}  
  
etc.
```

Methods that should not change the data are declared as const. This ensures that they *cannot* change the data.

Shallow versus Deep Copy

- The **assignment operator**, **copy constructor** and **destructor** must always be overloaded (written) for a class that has **pointer data (data on the heap)**.
[1]
 - If ***any*** of these 3 are needed, ***all 3*** are needed.
- To be safe, always write them but keep them empty for non-pointer data.
- This is because if you do not do so, the compiler will provide default versions for you.
- Such default versions may **not** do what you actually want them to do for pointer data. If there is no pointer data member, then the default versions are just fine – but see above concerning safety.
- For example, if you have a pointer in a class, the default versions would copy the value of the *pointer* itself, rather than make a copy of what the pointer is pointing to!
- The copying of a pointer instead of that to which it is pointing, is called a shallow copy. It results in one data being pointed to by more than one pointer.
- The copying of the contents of the memory to which it is pointing is called a deep copy.

Simple Pointer Class

- `class Pointer // simple illustration only – not complete to demonstrate what happens if care is not exercised in //design when the advice that is provided earlier is not followed.`
- `{`
- `public:`
- `Pointer () {m_ptr = NULL;} // nullptr is preferred instead of NULL`
- `~Pointer () {Clear ();}`
- `void Clear ();`
- `// Returns false if there is no memory available`
- `bool Set (int number);`
- `// friend shouldn't be here, but it is convenient for now to do convenient output.`
`Convert it non-friend and non-member as an exc.`
- `// a get method would be needed to make the conversion work.`
- `friend ostream& operator << (ostream &ostr, const Pointer &pointer);`
- `private:`
- `int *m_ptr; // it is an integer pointer. [1] [2]`
- `// Has pointer data, so copy constructor and assignment operator is also needed along with the destructor`
- `};`

- `//-----`
- `void Pointer::Clear ()`
- `{`
- `if (m_ptr != NULL)`
- `{`
- `delete m_ptr;`
- `}`
- `m_ptr = NULL;`
- `}`

- `//-----`
- `bool Pointer::Set (int number)`
- `{`
- `if (m_ptr == NULL)`
- `{`
- `m_ptr = new int; // “new” creates the memory space (heap) to store the number value`
- `}`
- `if (m_ptr == NULL) // no more heap memory available`
- `{`
- `return false;`
- `}`
- `else`
- `{`
- `*m_ptr = number; // copy the number value in the newly created heap memory`
- `return true;`
- `}`
- `}`

- //-----
- // is declared friend, so direct access to private data member
- // for debugging purposes **only**
- ```
ostream& operator << (ostream &ostr, const Pointer &pointer)
{
 ostr << "m_ptr is stored at location: " << &(pointer.m_ptr)
 << endl;
 ostr << "m_ptr points to location: " << pointer.m_ptr << endl;
 ostr << "contents of location is: " << *pointer.m_ptr << endl;

 return ostr;
}
```
- //-----

- **int main()**
- {
- Pointer ptr1;
- Pointer ptr2;
- 
- ptr1.Set (89);
- ptr2 = ptr1;
- 
- cout << "Pointer 1:" << endl;
- cout << ptr1;
- cout << endl;
- 
- cout << "Pointer 2:" << endl;
- cout << ptr2;
- cout << endl;
- 
- **return 0;**
- }

# Output From Test Program

**Pointer 1:**

```
m_ptr is stored at location: 0012FF70
m_ptr points to location: 00321E08
contents of location is: 89
```

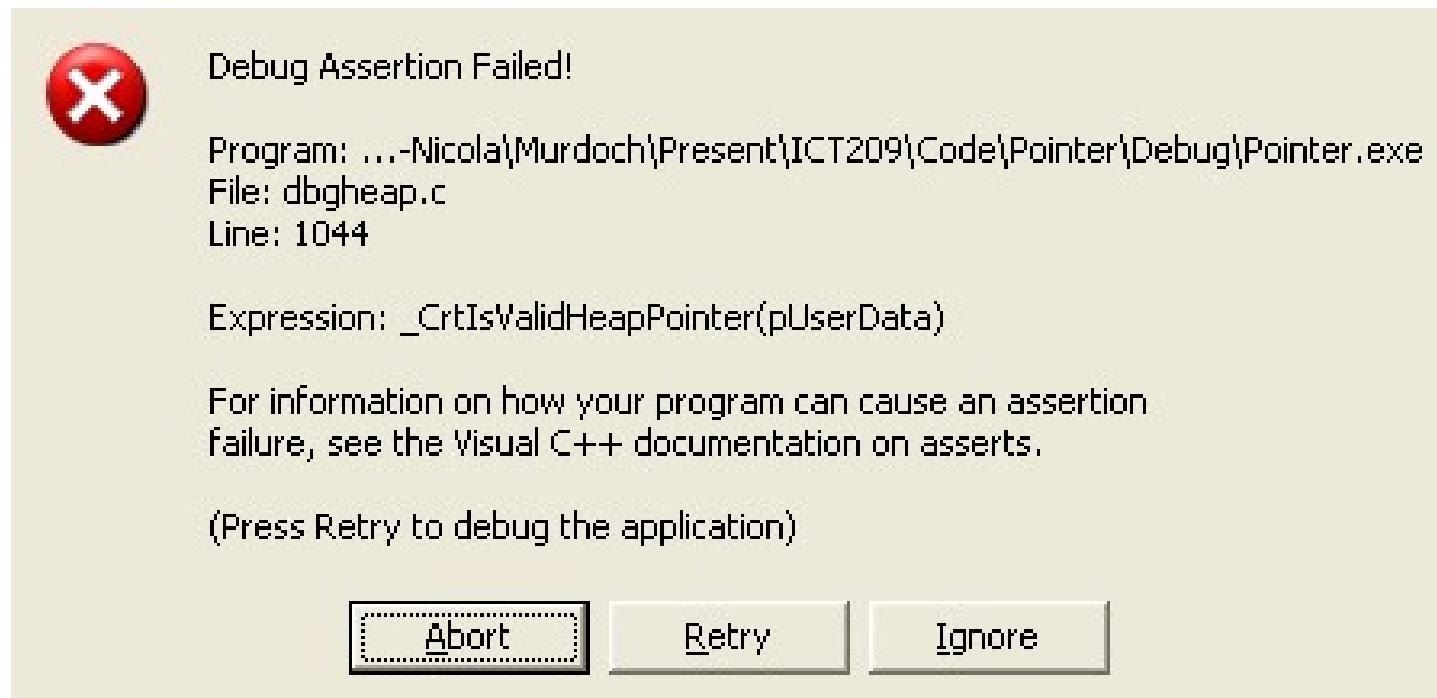
**Pointer 2:**

```
m_ptr is stored at location: 0012FF6C
m_ptr points to location: 00321E08
contents of location is: 89
```

The two Pointer  
objects point to the  
same location!

# The Destructor

- Destructors are actioned in the opposite order to the construction of objects.
- Therefore in the test program, pointer2 destructs, followed by pointer1.
  - But in this case, it does not matter as the problem exists either way.
- When pointer2 destructs, it releases the memory to which it points.
- Unfortunately, when pointer1 destructs it tries to do the same thing, so we get:

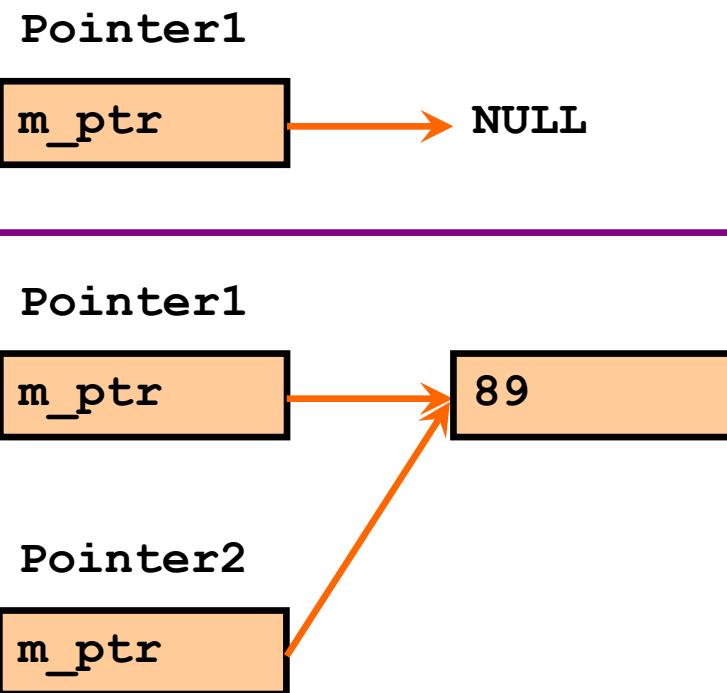


# Shallow Copy in Memory

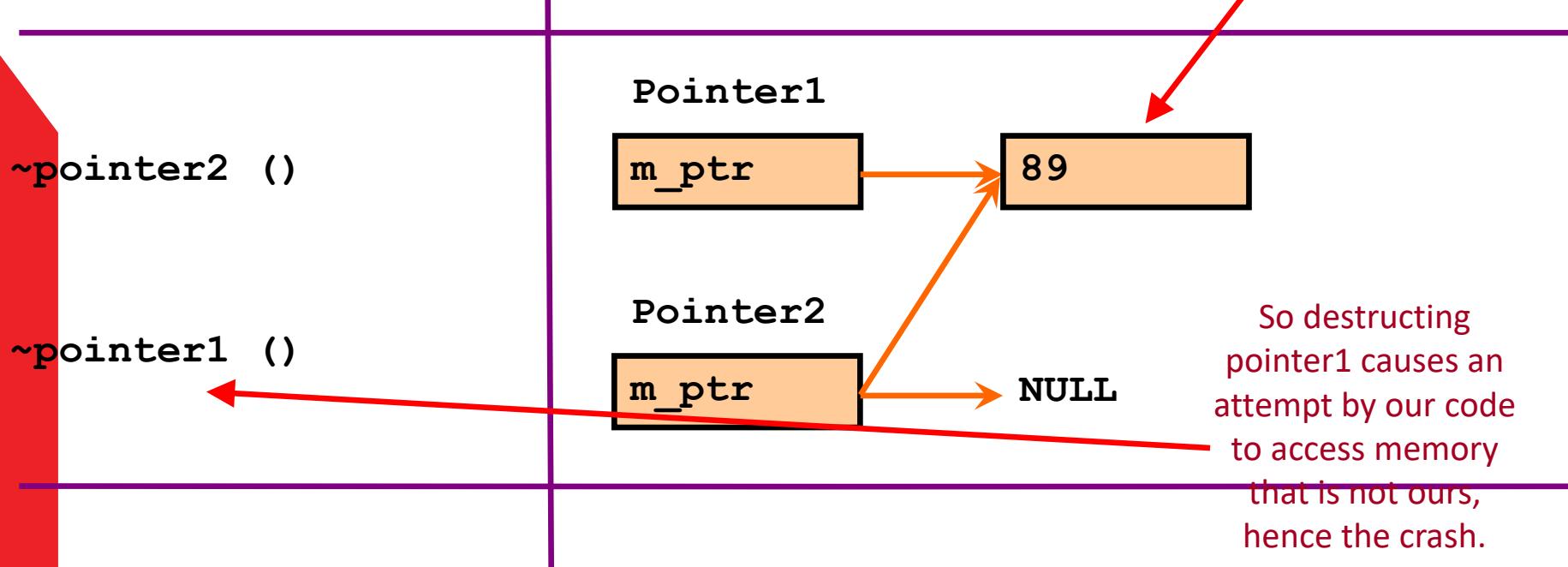
```
Pointer pointer1;
```

```
pointer1.Set(89);
```

```
pointer2 = pointer1;
```



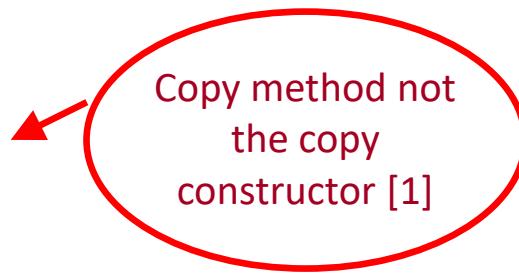
# Shallow Copy in Memory



# Preventing a Shallow Copy

- You can ensure a deep copy by
  - Writing a copy method;
    - Calling it from the assignment operator;
    - Calling it from a copy constructor.
- OR
  - Privatising the copy constructor;
  - Privatising the assignment operator;
  - Thereby preventing the compiler from creating default versions.

# Ensuring a Deep Copy

```
• class Pointer
• {
• public:
• Pointer () {m_ptr = NULL;} //prefer nullptr
•
• Pointer (const Pointer &initialiser); 
•
• ~Pointer () {Clear ();} // destructor prevents memory leaks
•
• void Clear ();
•
• bool Copy (const Pointer &rhs); // [1] should be private or protected 
•
• // Returns false if there is no memory available
• bool Set (int number);
•
• // Get method would be needed when converting to non-friend, non-member
•
• friend ostream& operator << (ostream &ostr, const Pointer &pointer); // convert to non-friend, non member
•
• Pointer& operator = (const Pointer &rhs); 
•
• private:
• int *m_ptr;
• };
• }
```

# Copy Constructor

- //-----
- Pointer::Pointer (**const** Pointer &initialiser)
- {
- m\_ptr = NULL; // Set method needs this, constructor sets to null
- **Copy (initialiser);**
- }

Copy () is then called, but note that we cannot return a value from a constructor, so if Copy () fails, we have no way of knowing in code.

# Copy Method

- //-----
- ```
bool Pointer::Copy (const Pointer &rhs)
{
    if (rhs.m_ptr != NULL) // what happens if you don't check?
    {
        return Set (*(rhs.m_ptr)); // with rhs int data value
    }
    else
    {
        return false;
    }
}
```

Overloaded Assignment Operator

- //-----
- Pointer& Pointer::operator = (const Pointer &rhs)
- {
- Copy (rhs);
- return *this;
- }

‘this’ is a pointer to the object itself.

Therefore ‘*this’ is the contents of the object.

Returning it fulfills the requirements of the assignment operator.

Like the constructor, it simply uses the Copy () method.

It is also similarly dangerous!

```
• int main()
• {
•     Pointer ptr1;
•     Pointer ptr2;
•
•     ptr1.Set (89);
•     ptr2 = ptr1;
•
•     Pointer ptr3 (ptr1);
•
•     cout << "Pointer 1:" << endl;
•     cout << ptr1;
•     cout << endl;
•
•     cout << "Pointer 2:" << endl;
•     cout << ptr2;
•     cout << endl;
•
•     cout << "Pointer 3:" << endl;
•     cout << ptr3;
•     cout << endl;
•
•     return 0;
• }
```

These two statements now lead to deep copies of ptr1

Output From Test Program

Pointer 1:

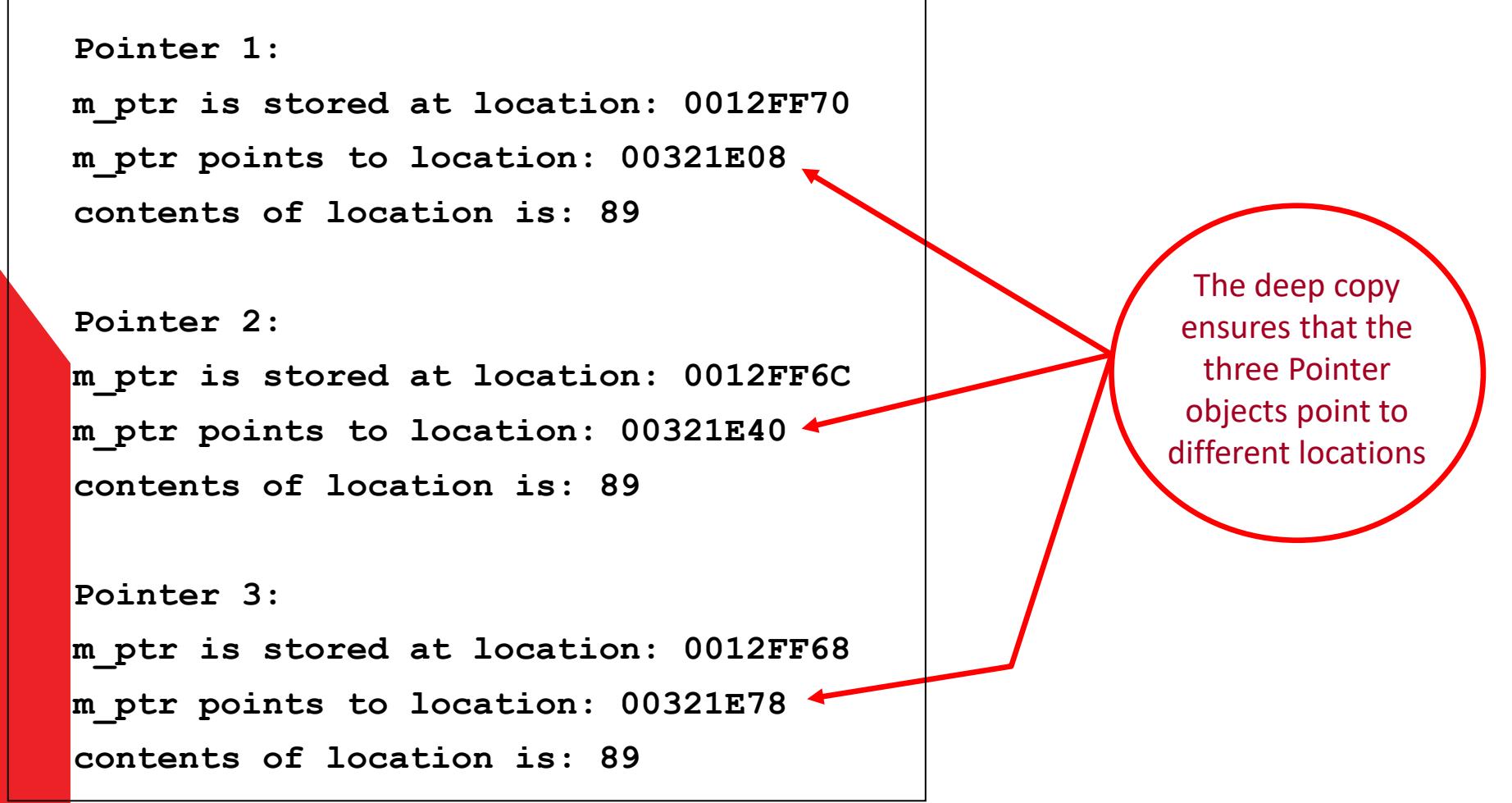
```
m_ptr is stored at location: 0012FF70  
m_ptr points to location: 00321E08  
contents of location is: 89
```

Pointer 2:

```
m_ptr is stored at location: 0012FF6C  
m_ptr points to location: 00321E40  
contents of location is: 89
```

Pointer 3:

```
m_ptr is stored at location: 0012FF68  
m_ptr points to location: 00321E78  
contents of location is: 89
```



The deep copy ensures that the three Pointer objects point to different locations

Preventing Default Versions [1]

```
• class Pointer
• {
•     public:
•         Pointer () {m_ptr = NULL;}
•         ~Pointer () {Clear();}
•
•         void Clear ();
•         bool Copy (const Pointer &rhs) {return Set (*rhs.m_ptr);}
•
•         // Returns false if there is no memory available
•         bool Set (int number);
•
•         friend ostream& operator << (ostream &ostr, const Pointer &pointer);
•
•     private:
•         int *m_ptr;
•
•         Pointer& operator = (const Pointer &rhs) {return *this;}
•         Pointer (const Pointer &initialiser) {}; //=delete [2]
•     };
}
```

Privatised declarations prevent outside code using neither of the assignment operator nor the copy constructor. [2]

Preventing Default Versions

```
• class Pointer
• {
•     public:
•         Pointer () {m_ptr = NULL;}
•         ~Pointer () {Clear();}
•
•         void Clear ();
•         bool Copy (const Pointer &rhs) {return Set (*rhs.m_ptr);}
•
•         // Returns false if there is no memory available
•         bool Set (int number);
•
•             // convert to non-friend, non-member. Get method would be needed.
•             friend ostream& operator << (ostream &ostr, const Pointer &pointer);
•
•     private:
•         int *m_ptr;
•
•         Pointer& operator = (const Pointer &rhs) {return *this;} [1]
•         Pointer (const Pointer &initialiser);
•     };
}
```

The test program
will now be
prevented from
compiling.

Some Final Points

- Mutator/Set methods should set one piece of data only and should return a boolean to indicate success or failure.
- Call other methods rather than re-write code.
- Never do output in any method other than an output method. Data Structure classes do not have an output method. The use accessor/get methods.
- A data class should not do input from file/keyboard or output to screen/file.
- Make every class you write *minimal*: only include those methods that you know you need. See earlier notes about what constitutes minimal.

Readings

- Textbook: Chapter on Classes and Data Abstractions.
- Textbook: Chapter on Pointers, Classes, Virtual Functions, Abstract classes, and Lists: *Section* on Shallow versus Deep Copy and Pointers; *Section* on Classes and Pointers: Some peculiarities.



Murdoch
UNIVERSITY

Data Structures and Abstractions

Parameters and Inheritance

Lecture 14



Time Wasting Code

- // Pointless program that does nothing!
- 1: int main()
- 2: {
- 3: for (int index = 0; index < 10; index++)
- 4: {
- 5: Light light; // constructor used
- 6: light = InputLight();
- 7: }
- 8:
- 9: cout << endl;
- 10: return 0;
- 11: }

- 12: Light InputLight ()
- 13: {
- 14: Light light; // constructor used
- 15:
- 16: float radius;
- 17: cout << "Enter radius of light in centimeters: ";
- 18: cin >> radius;
- 19: light.SetRadius(radius);
- 20:
- 21: string colour;
- 22: cout << "Enter colour of light: ";
- 23: cin >> colour;
- 24: light.SetColour (colour);
- 25: return light; // copy constructor used
- 26: }

How Many Constructors?

Index

0

5: Light light;

Construction

Count

1

How Many Constructions?

Index

0

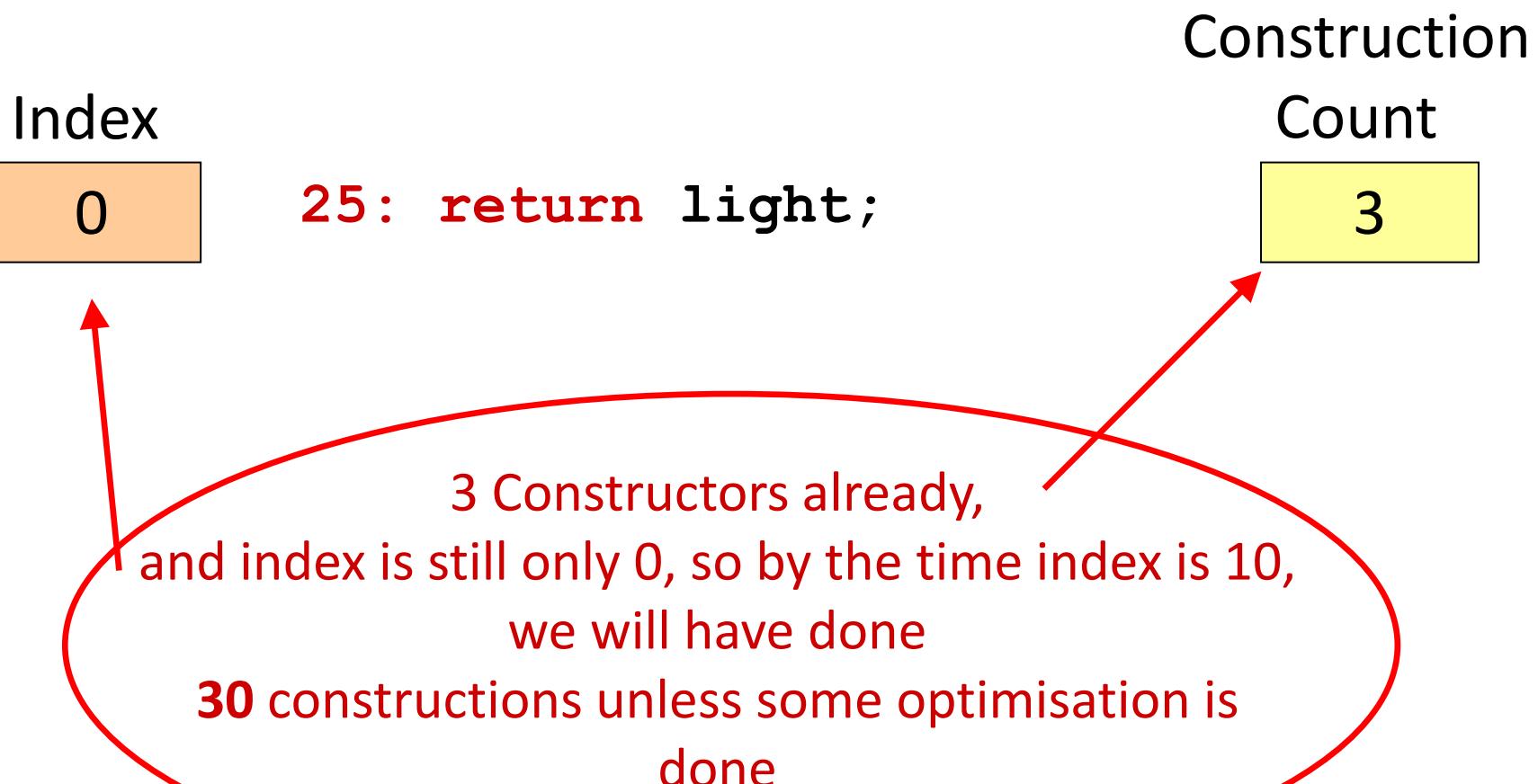
14: Light light

Construction

Count

2

How Many Constructors?



Good Code (refactored)

```
• // Pointless program that does nothing!
• int main()
• {
•     Light light;
•     for (int index = 0; index < 10; index++)
•     {
•         InputLight (light);
•     }
•     cout << endl;
•     return 0;
• }
```

The constructor is now outside the loop.

The data is now returned as a parameter rather than function-wise.

```
• void InputLight (Light &light)
• {
•     float radius;
•     cout << "Enter radius of light in centimeters: ";
•     cin >> radius;
•     light.SetRadius(radius);
•
•     string colour;
•     cout << "Enter colour of light: ";
•     cin >> colour;
•     light.SetColour(colour);
• }
```

The data is now returned as a parameter rather than function-wise.

Therefore we do not need a local variable.

The 30 constructions is now reduced to only one!

Summary

- Construction of an object takes time.
- Therefore the more constructions, the slower the code.
- When returning an object by value function-wise, there is a hidden construction (copy construction) as the data is transferred back to the calling function.
- This extra construction time cost is exacerbated if it is placed within a loop with a local variable.
- Objects passed-by-value into a function also cause an extra construction – copy constructor used.

Rules

- It is important to follow the correct rules for parameter passing and function returning.
- The rules are designed to make your code as efficient and bug-resistant as possible.
- Failure to stick to these rules may cost you marks in assignments. You would also have spent time trying to fix poor code, so it is not worth ignoring the rules.

- *Rule: [1]* Simple types (int, float etc) are passed by either value or non const reference or returned function-wise:
 - Nothing is to be returned:
`void DoSomething (float num);`
 - Change expected to the variable to be returned:
`void DoSomething (float &num);`
 - Something is expected to be returned:
`float DoSomething (float num);`
- *Rule: Objects* are always passed by reference
 - No change expected to light:
`void DoSomething (const Light &light);`
 - Change expected to light:
`void DoSomething (Light &light);`

Inheritance

- Inheritance tends to get overused and badly used.
- However, there are occasions when it is both correct and useful.
- **Inheritance is correct to use when:**
 1. the derived class (sub-class, child) “is a” parent class (super-class),
 2. the derived class requires all the data declared in the parent,
 3. the derived class uses every method defined in the parent.

Examples

- The PatternLight described in one of the earlier lectures is a good example of correct use of inheritance:
 - PatternLight *is a* Light
 - PatternLight requires **m_colour**, **m_radius** and **m_on**
 - PatternLight will use all of Light's methods.
- A poor example would be Square inheriting from Rectangle:
 - Square *is a* Rectangle
 - BUT, Square does *not* need **m_width**, which would have been defined in Rectangle.

Protected Data

- For the Light class, we made all data private.
- Private data is protected from absolutely everything, including derived classes.
- Therefore when you derive a class, you need to alter the parent class so that the data is *protected* rather than private if you want derived classes to access parent data.
- Protected data is protected from view by the outside world, but available to derived classes.
- One can make all data protected as a rule, and then never have to go back and make changes.
[1] But this is terrible. Only classes that are meant to be derived from should have “protected” specified.

Constructors and Destructors

- When you construct a class that is derived from another class, the default constructor of the parent class is automatically run before the constructor of the derived class. [1]
- However the **destructor of the parent class is not automatically run**.
- To ensure that it *is* automatically run, you need to add the '**virtual**' keyword in front of it (parent destructor) in the header file:
virtual ~Light () ;
- Destructors are run in *reverse* order: the child class and then the parent class.

Virtual Methods

- If a method in the parent class is to be over-ridden in the child class, then it too is declared as virtual: [1]

```
virtual void DoSomething () ; // parents
```

- If the child class wishes to access the parent class' version, then it uses the scope resolution operator:

```
void Child::DoSomething ()  
{  
    parent::DoSomething () ; // call parent's  
    version.  
}
```

Required Changes to the Light Class

```
• class Light
• {
•     public:
•         Light () {Clear();}
•         virtual ~Light () {};
•
•         virtual void Clear ();
•         //...
•
•     protected:
•         // Any string is acceptable, we shall assume it is a colour
•         string m_colour;
•         // In centimetres
•         float m_radius;
•         bool m_on;
•     };
• }
```

The destructor becomes 'virtual'

The Clear() operator becomes virtual

Data becomes 'protected' rather than private.

```
• // A light that shines through a cutout giving it a particular shape
• //
• // Version
• // 01 - Nicola Ritter
• //
• //-----
```



```
• #ifndef PATTERN_LIGHT
• #define PATTERN_LIGHT
```



```
• //-----
```



```
• #include "../Light/Light.h"
```



```
• //-----
```



```
• // Available shapes
• //-----
```



```
• const int NO_SHAPE = 0;
• const int LEFT_ARROW = 1;
• const int RIGHT_ARROW = 2;
• const int MAX_SHAPE = 2;
```

The Light header file must be included.

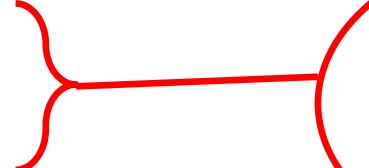
You can list as many shapes as you want, but make sure that MAX_SHAPE is changed to match the highest number

```
• //-----  
• class PatternLight : public Light [1]  
• {  
• public:  
•     PatternLight () {Clear();}  
•     PatternLight (const PatternLight &plight);  
•     virtual ~PatternLight () {};  
•  
•     void Clear ();  
•  
•     bool SetShape (int shape);  
•     int Get () const {return m_shape;}  
•  
•     friend ostream& operator << (ostream &ostr, const PatternLight &light); [2]  
•     PatternLight & operator = (const PatternLight &plight);  
•  
• private:  
•     int m_shape;  
• };  
• //-----  
• #endif
```

The Clear method overrides those in the Light class

Set and Get methods are only required for this class' attributes.

- **void PatternLight::Clear ()**
- {
- Light::Clear();
- m_shape = NO_SHAPE;
- }



PatternLight calls the
Clear() from the Light
parent, before
initialising its own
attributes.

- **bool** PatternLight::SetShape (int shape)
- {
- **if** (shape >= NO_SHAPE && shape <= MAX_SHAPE)
- {
- m_shape = shape;
- **return** true;
- }
- **else**
- {
- **return** false;
- }
- }

```

• ostream& operator << (ostream &ostr, const PatternLight &light)
• {
•     ostr << static_cast<Light>(light);
•
•     if (light.m_on && light.m_shape != NO_SHAPE)
•     {
•         ostr << ", showing ";
•         switch (light.m_shape)
•         {
•             case LEFT_ARROW:
•                 ostr << "left arrow";
•                 break;
•             case RIGHT_ARROW:
•                 ostr << "right arrow";
•                 break;
•         }
•     }
•
•     return ostr;
• }

```

The `static_cast` tells the compiler to redefine `light` as a `Light` instead of a `PatternLight`.

This line of code, therefore, runs the output code in the parent class.

Therefore all this method has to do is output information based on this class' attributes

Make sure you *only* use a `static_cast` when there is an inherit relationship between the two.

Readings

- Textbook: Chapter on Classes and Data Abstractions.
- Chapter on Inheritance and Composition.



Murdoch
UNIVERSITY

Data Structures and Abstractions

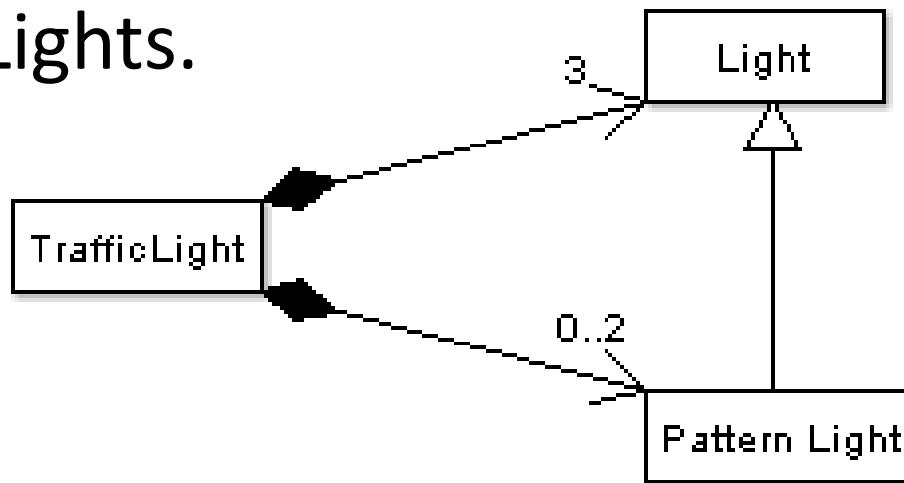
Composition, Aggregation & Templates

Lecture 15



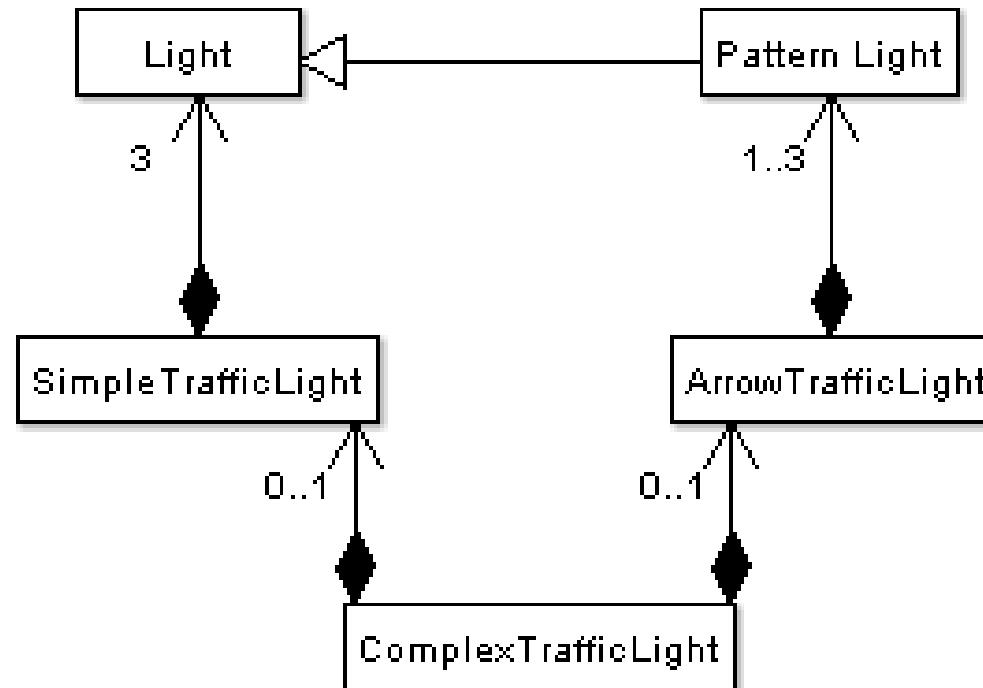
Composition

- Composition is where one class has a data member that is an object of another class.
- The example given in Lecture 11, was TrafficLight, which had three Lights and 0..2 PatternLights.



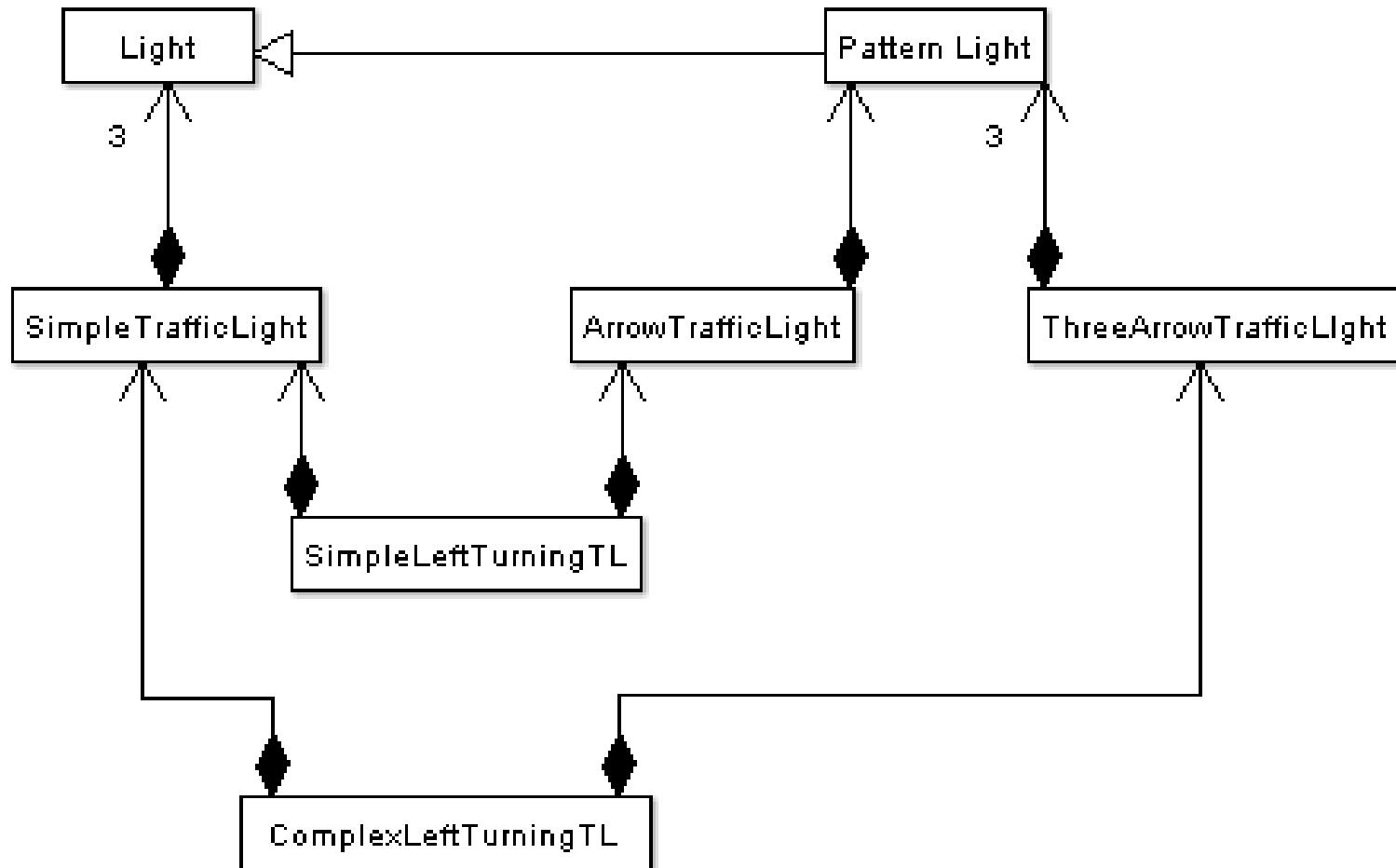
Design Change

- Lets say that after thinking about the problem a design change was needed as shown below.



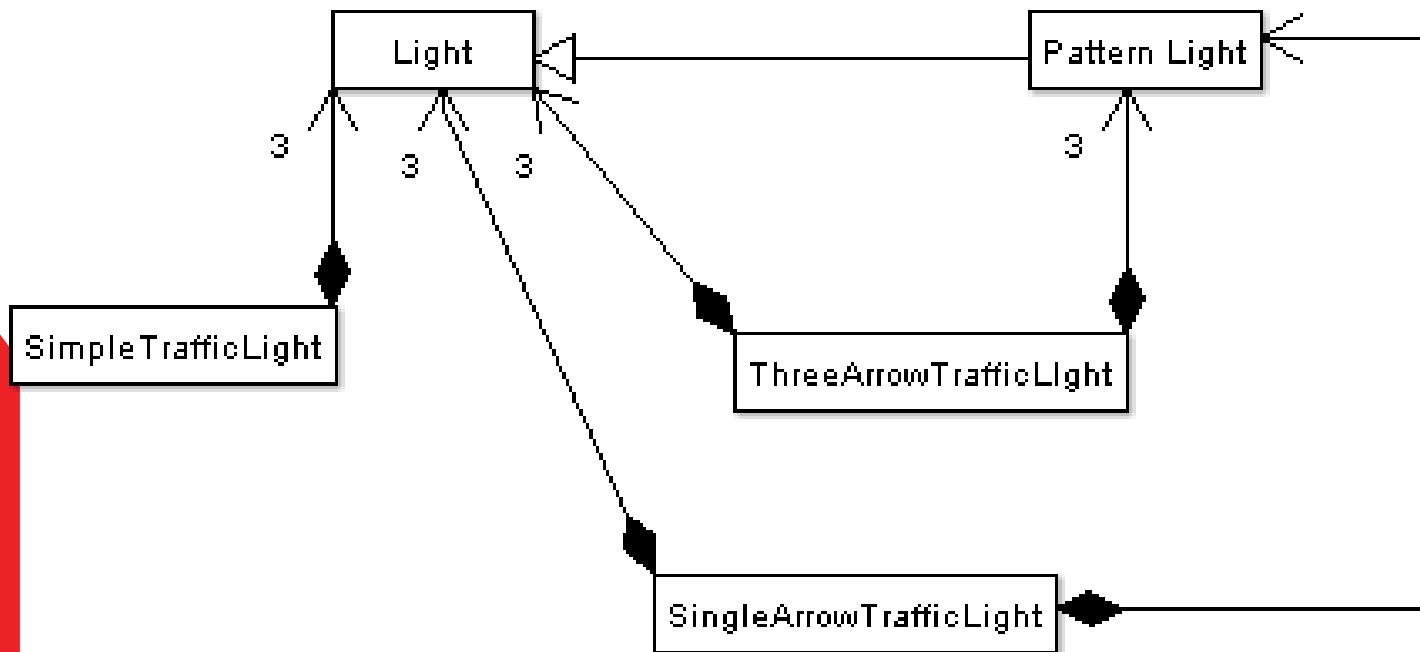
Another Design Change

- And then when working on the algorithm, it was realised that the behaviour changed depending on the number of lights in a traffic light.
- This would have meant that within the code there would be lots of `if` statements to do with how many of each type of light.
- This was a sure sign that the design was still incorrect, so another change was made to the design.



Yet Another Design Change

- However, after a while I hit problems again.
- The composition seemed forced and the whole thing very complicated: a sure sign it was still incorrect.
- So I went out for a drive to look at traffic lights in action.
- I quickly realised that a complex traffic light was not composed of a simple traffic light and a three arrow traffic light, it was actually composed of three single and three pattern lights.
- So my ‘final’ design was:



Design Changes Continued

- Of course there would be lots more types of traffic light than just these. [1]
- And if I was really coding the *whole* thing, I might well decide that my original design (or something else entirely) was more correct.
- Which is the wonderful advantage of software engineering over conventional engineering: design need not be static but the most important lesson is when starting on a problem solving task, find out what is the real problem!!!
 - Don't try to just imagine what the problem is going to be.
 - Do some "leg work", talk to end users, as changes in design (even in software engineering) has costs associated with it.
- Remember:
 - Code incrementally.
 - If everything is becoming too complicated: you have almost certainly stuffed up the design.
 - Don't be afraid to change your design. Implementing a wrong design will make the solution useless.
 - Don't be afraid to question the design of others.
 - Refactor incrementally.
 - Test everything after every change.

Finite State Machines

- The traffic light classes are examples of Finite State Machines.
- An FSM has a limited set of states that are visited one after the other.
- It is not possible to have two states at once: you cannot have both a red and green light showing in a normal FSM. There are fuzzy FSMs but these are outside the scope of this unit.
- FSM are used in simulation and modelling of many industrial and mechanical processes. Even the compiler you are using to compile your code uses state machines.
- A single **Change ()** method replaces all the **Set ()** methods.
- A single **StateIs ()** method replaces all **Get ()** and output methods.
- The term **Initialise ()** is used rather than **Clear ()**, as it is only usually called at the start.

- // Constants.h
- // Required by several classes
- //-----
- #ifndef CONSTANTS
- #define CONSTANTS
- // Traffic light colours
- const int RED_LIGHT = 0; [1]
- const int ORANGE_LIGHT = 1;
- const int GREEN_LIGHT = 2;
- // FSM error state
- const int ERROR_STATE = -1;
- // Size of a traffic light
- const float TL_RADIUS = 12.5; // cm
- #endif

Constants that are required by multiple classes are put in a header file of their own.

- // SimpleTrafficLight.h
- // Comments and includes etc up here as per normal – use doxygen comments instead
- // friend operator used for debugging. Design does not require it.
- //-----
- const int STL_NUMBER = 3;
- //-----
- class SimpleTrafficLight
- {
- public:
- SimpleTrafficLight () {Initialise();}
- ~SimpleTrafficLight () {};
- // Initialise the class
- void Initialise ();
- // Change the light to the next state
- bool Change ();
- // Output the state – for debugging/demo purposes only.
- friend ostream& operator << (ostream &ostr, const SimpleTrafficLight &light);

- **private:**
- **int m_state;**
- **vector<Light> m_lights; [1]**
- **// Clear all old data**
- **void Clear();**
- **// Set the light sizes and colours**
- **void InitialiseLights();**
- **};**
- **#endif**

- // SimpleTrafficLight.cpp
- // Comments and includes as per normal
- //-----
- void SimpleTrafficLight::Initialise ()
- {
- Clear ();
- // Add the correct number of lights to the vector
- Light light;
- for (int index = 0; index < STL_NUMBER; index++)
- {
- m_lights.push_back (light); // think about this. Is it the same light?
- }
- InitialiseLights ();
- }

- *//-----*
- **void SimpleTrafficLight::Clear ()**
- {
- m_lights.clear();
- m_state = ERROR_STATE;
- }
- *//-----*

- **void** SimpleTrafficLight::InitialiseLights()
- {
- *// Set the radii of the lights*
- **for** (**int** index = 0; index < STL_NUMBER; index++)
- {
- m_lights[index].Set(TL_RADIUS);
- }
- *// Set the colours*
- m_lights[RED_LIGHT].Set("red");
- m_lights[ORANGE_LIGHT].Set("orange");
- m_lights[GREEN_LIGHT].Set("green");
- *// Switch the red light on*
- m_lights[RED_LIGHT].Switch();
- *// Set the state*
- m_state = RED_LIGHT;
- }

```
•     bool SimpleTrafficLight::Change()
•     {
•         switch (m_state)
•         {
•             case RED_LIGHT:
•                 m_lights[RED_LIGHT].Switch();
•                 m_lights[GREEN_LIGHT].Switch();
•                 m_state = GREEN_LIGHT;
•                 break;
•             case GREEN_LIGHT:
•                 m_lights[GREEN_LIGHT].Switch();
•                 m_lights[ORANGE_LIGHT].Switch();
•                 m_state = ORANGE_LIGHT;
•                 break;
•             case ORANGE_LIGHT:
•                 m_lights[ORANGE_LIGHT].Switch();
•                 m_lights[RED_LIGHT].Switch();
•                 m_state = RED_LIGHT;
•                 break;
•         }
•         return (m_state != ERROR_STATE);
•     }
```

- **// As an exercise convert this to non-friend, non-member operator**
- **ostream& operator << (ostream &ostr, const SimpleTrafficLight &light)**
- **{ [1]**
- **for (int index = 0; index < STL_NUMBER; index++)**
- **{**
- **string colour;**
- **light.m_lights[index].Get(colour);**
- **if (index == light.m_state)**
- **{**
- **ostr << "O " << colour << endl;**
- **}**
- **else**
- **{**
- **ostr << "o" << endl;**
- **}**
- **}**
- **return ostr; [2]**
- **}**

Simple Unit Test Program

```
• int main() // not a complete unit test until you have code to test each method.  
• {  
•     SimpleTrafficLight light;  
•  
•     cout << "Each time you press <Enter> the lights will change,"  
•         << "use <Ctrl>-C to end the program." << endl;  
•  
•     while (true)  
•     {  
•         cout << light << endl;  
•         light.Change();  
•         getchar(); // clunky!!  
•     }  
•  
•     cout << endl;  
•  
•     return 0;  
• }
```

Aggregation

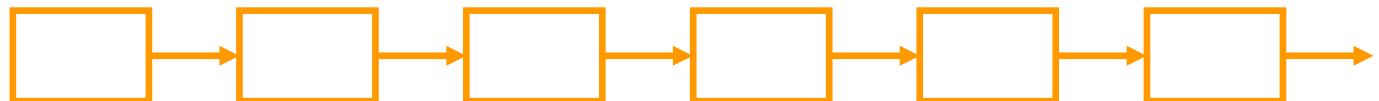
- Aggregation is used when a class has an attribute that is the object of another class, but it does not have control over the construction and destruction of that object.
- A common use of aggregation occurs in Windows programming, where many objects may want to refer to the current window, however none of them have the power to delete (close/destruct) the window.
- Similarly, a Unit class would be associated with a lecturer and students, but would not control them, so a Unit offering would have an aggregation of a Unit, lecturers and students, rather than a composition of them. Unit offering would be the class that contains all of these aggregations.
- Aggregation necessitates using an attribute that is **either** an **index**, **reference** or **pointer** to the aggregated object.
- Does aggregation actually exist at all, or is the class actually composed of a *pointer* or *reference*?
- In this unit we will not worry about the why's or wherefore's we will simply use and talk about aggregation as described above.

Linked Lists

- A good example of aggregation is what occurs in a linked list.
- A linked list is exactly what it sounds like: each piece of data is combined with a link (pointer) to the next piece of data. This combination is called a *node*.
- In that situation the node class is composed of the data, but aggregates the next node.
- This is because if we delete/remove a node, it does not automatically delete the following node.
- We will cover lists again in a later lecture. For this topic, you need to know the basics.

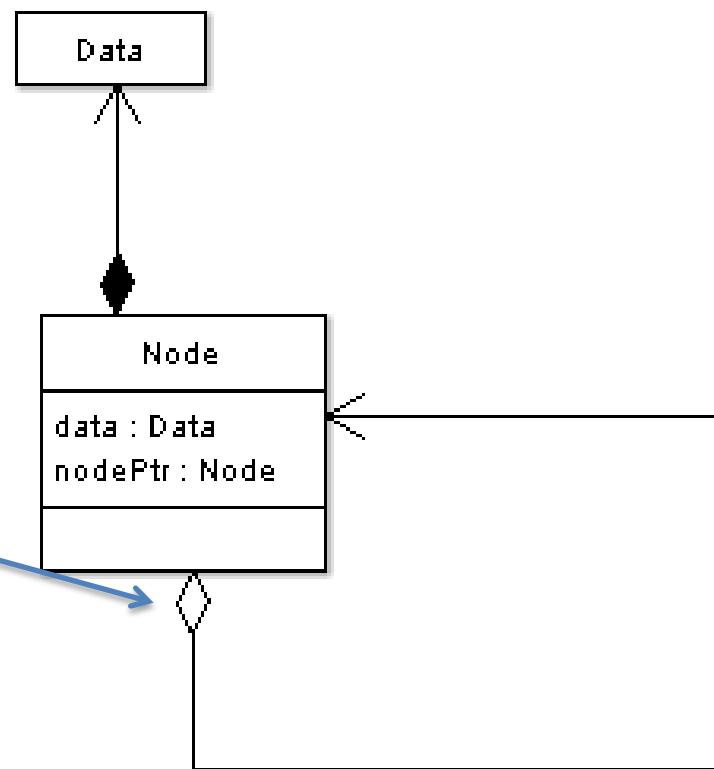
Linked List Diagrams

Abstract View



UML Diagram

When you have this notation, is Node responsible for deleting itself?



- In C++, a simplified node class that stores a single integer piece of data, might look like this:
 - `class Node [1] // class or struct – think carefully`
 - `{`
 - `public:`
 - `Node () {m_next = NULL;} // or nullptr. Always initialise to null`
 - `~Node () {} // [2]`
 - `Node *GetNext ();`
 - `void SetNext (Node *next) {m_next = next;}`
 - `int GetData () {return m_data;}`
 - `void SetData (int data) {m_data = data;}`
 - `private:`
 - `int m_data; // data is strongly associated with Node – see UML`
 - `Node *m_next; // aggregation. Would the destructor delete this?`
 - `};`

Templates <>

- The Node class is almost identical no matter what type of data is stored within it.
- Therefore, rather than re-write it every time we want to store a different data type, we use a *template*.
- Templates are *descriptions* of types which have to be instantiated with a particular type at run time.
- We have already used them when using the STL.
- A template node class would look like this:

```
• template <class DataType> [1]
  • class Node
  • {
  • public:
  •     Node () {m_next = NULL;}
  •     ~Node () {}

  •     Node *GetNext ();
  •     void SetNext (Node *next) {m_next = next;}

  •     void Data (DataType &data) {data = m_data;}
  •     void SetData (const DataType &data) {m_data = data;}

  • private:
  •     DataType m_data; // same idea as before about relationship
  •     Node *m_next;
  • }
```

We tell the compiler it is a template. The type is now a parameter.

We use DataType to replace 'int' everywhere

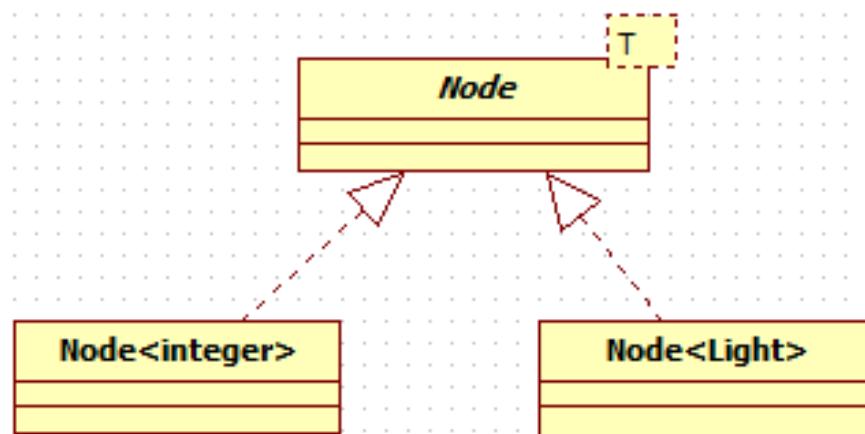
- Within our program, we then instantiate a node in the same way as with the STL:

```
typedef Node<int> IntNodeType; // IntNode is a type
IntNodeType intNode;
```

- IntNodeType is a type; in this case an integer Node class.
- The template Node can't contain anything as T is not bound to a type. Once bound to a type, it is *realised*, and can be used in an application.
- There are limitations to templates:
 - They can only be used where all the methods and attributes are the same for every class.
 - They require all the code to go in the header file or else the code file has to be included! See textbook chapter on, Overloading and Templates for further discussion.
 - They can make the code *very* difficult to debug.
- They should be avoided except for very simple classes with only a few, clearly defined methods and attributes – generic classes.

In UML

- The box (with T) should have a **dashed border**. Arrows are **diamond** shaped. Lines are dashed. (Realisation [1] created in *StarUml* tool)



- The Realised types `Node<integer>` and `Node<Light>` can contain data.
- The template `Node` cannot contain data.

Exercise

- How would you the ***Law of Demeter [1]*** be applied when applied to objects that are composed of other objects (composition or aggregation)?
- Would there be situations where it would make sense to violate this law?

Readings

- Textbook: Chapter on Classes and Data Abstractions.
- Chapter on User-Defined simple data types, Namespaces and the string Type.
- Chapter on Inheritance and Composition.
 - You should go through the Programming example: Grade Report in the chapter on Inheritance and Composition.
- Entire chapter on Pointers, Classes, Virtual Functions, Abstract classes, and Lists.
- Chapter on Overloading and Templates.



Murdoch
UNIVERSITY

Data Structures and Abstractions

Abstract Classes

Lecture 16



Background – Data Type

- Data Type
 - Has a name
 - Has a set of values
 - Has a set of operations on these values
- Data Types
 - “atomic” data types
 - values are “not” decomposable, e.g. Integer
 - Data Structures
 - Values are decomposable
 - Values are related, e.g array of integer

Background – Data Type

- Abstraction of Data Types [\[1\]](#)
 - Abstract Data Type (ADT)
 - Product of our imagination
 - Only essential properties – no details of implementation
 - Virtual Data Type (VDT)
 - Exists on a virtual processor – e.g. Programming language
 - Physical Data Type (PDT)
 - Exists on the machine – the machine representation
- VDTs implement ADTs
- PDTs implement VDTs

Background – Data Type

	Abstract	Virtual	Physical
Atomic	Number of chairs	C++ or Java .. etc integer	“series” of bits
Structured	List of chairs	C++ or Java .. etc Array of classes or structs	“series” of bytes

At the Abstract level, you do **not** think of the Programming Language.

When you are doing OO design, think at the Abstract level. You want your classes (at the virtual level of abstraction) **mimicking the Abstract level.** [\[1\]](#)

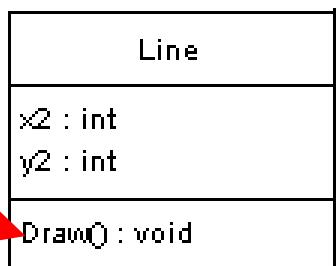
Abstract Classes

- Do not forget the big picture on what is “Abstraction” covered earlier. When we are considering Abstract Classes in C++, we are considering the virtual level of abstraction. The fact that the word “virtual” is used – see later – can be helpful but can also be a source of confusion. [1]
- When one class inherits from another class, a method might be replaced. In the parent class the method is designated a virtual method:
virtual DoSomething () ; // polymorphic method
- If the method in the parent class is to be replaced, but is not actually to be defined in the parent class, then the virtual method must become a ‘pure virtual method’:
virtual DoSomething () = 0; // parent doesn't have a code body
 - Any class that contains pure virtual methods is—by default—an abstract (pure virtual) class: it cannot ever be instantiated as an object because there is missing code body.
- In UML, virtual classes are indicated by using italics for the class name, and the relationship of the derived classes is called a ‘realisation’. [2]
- In C++ realisations are implemented using inheritance in the same way as are derivations but with **dashed** line.

Pure Virtual Classes in UML

The Draw() method is overridden for each different type of ScreenObject

Realisations have similar arrowheads to derivations, but a dashed line.



virtual (abstract) classes are shown in italics [2]

In StarUML, this is in italics as well but other tools might not do it, so work around by saying void [1]

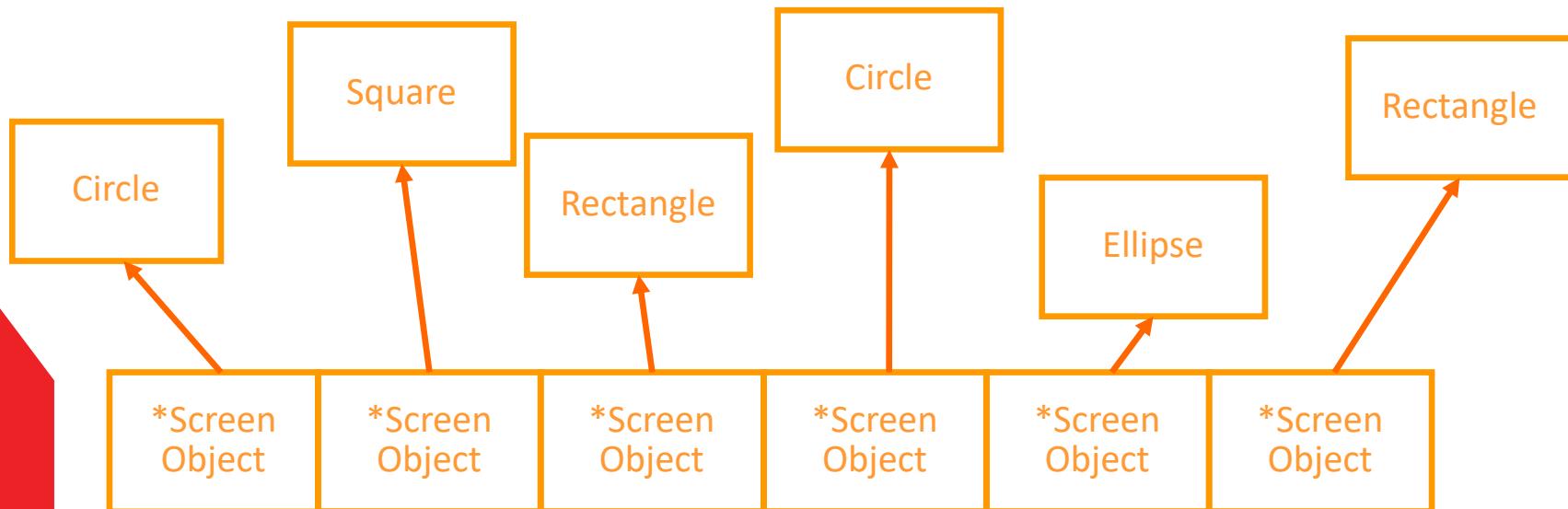
Note that Set and Get methods not shown in this UML diagram.
Protection is not shown either.

Uses of Abstract Classes

- One of the most common reasons for having an Abstract Parent class is so that different things can be grouped together in a single container. **Polymorphism** is used to distinguish the behaviour.
- For example, in a drawing program you need to have some kind of list of all the objects currently part of the drawing. As all of the objects can be drawn, the draw method is declared virtual in the parent. **This is one of the conditions needed for polymorphism to occur in C++.**
- And you need to be able to iterate through this list when drawing, saving, printing etc. **You will need to access the contained object via the parent pointer** (or reference). **This is the other condition for polymorphism to occur.**
- However strongly typed languages don't usually support an array (or list) of disparate things: **[1]**



- However, in C++ what you *are* allowed to do is have an array/list of *pointers* to disparate objects:



- When you iterate through the list, the program calls the `*ScreenObjects Draw()` methods, which are different for each of the classes as the determination of which `Draw()` to use gets delayed until run-time. **Polymorphism** is occurring [1].

- // Shape.h
- // A base class for drawing shapes.
- // Version
- // 01 - Nicola Ritter
- // 02 – smr – **see actual code in the Realisation project for this lecture**
- // includes additional explanation
- // IDENTIFY ALL DESIGN ISSUES
- //-----
- #ifndef SHAPE
- #define SHAPE
- #include <iostream> // should this #include be here?
- #include <string>
- using namespace std; // **should this exposure happen**
- //-----
- const float ASPECT_RATIO = 12.0/8.0; // [1]
- //-----

Characters in a
DOS box are
usually 12x8
pixels [1]

```
• // Read this together with the actual code in Realisations project
• class Shape
• {
• public:
•     Shape() {m_height = 0;}
•     virtual ~Shape () {}; // designed for inheritance, so virtual destructor
•
•     virtual void Input (); // virtual needed for polymorphism – see actual code
•     virtual void Draw () const = 0;
•
• protected:
•     int m_height;
•     string m_description;
• };
•
• //-----
• #endif
```

A pure virtual method,
therefore this is
an abstract class.
[1]

Attributes are protected
not private, so that
derived classes can access
them.

- `#include "Shape.h"`
- `//-----`
- `void Shape::Input ()// code for illustration only`
- `// I/O makes the class have reduced usage. [1]`
- `{`
- `cout << "Enter " << m_description << " height: ";`
- `cin >> m_height;`
- `}`
- `//-----`

If `m_description` is given a different value by each derived class, then this output will inform the user about the type of shape

- // Square.h
- // Version
- // 01 - Nicola Ritter
- //

- #ifndef SQUARE
- #define SQUARE

- //

- #include "Shape.h"

Need to include parent header

- //

- class Square : public Shape
- {
- public:

Derived from Shape

- Square() {m_description = "square";}
- virtual ~Square () {};

Need to initialise description

- virtual void Draw () const; // was declared pure in Shape, so this is needed

- private:
- // nothing here
- };

- #endif

Draw() method has to be defined as it was not defined in Shape.

```
• // Square.cpp

• #include "Square.h"

• //-----

• void Square::Draw () const
{
    for (int row = 0; row < m_height; row++)
    {
        for (int col = 0; col < m_height * ASPECT_RATIO; col++)
        {
            cout << '*';
        }
        cout << endl;
    }
    cout << endl;
}

• //-----
```

Ensures it will look like a square on screen.

- // Triangle.cpp
- #include "Triangle.h"
- //
- void Triangle::Draw () const
- {
- for (int row = 0; row < m_height; row++)
- {
- for (int col = 0; col < row+1; col++)
- {
- cout << '*';
- }
- cout << endl;
- }
- cout << endl;
- }
- //

Triangle.h is
almost
identical to
Square.h

For Triangles we
don't care
about the
aspect ratio

- `class Rectangle : public Shape`
- `{`
- `public:`
- `Rectangle();`
- `virtual ~Rectangle () {};`
- `virtual void Draw () const;`
- `virtual void Input ();`
- `private:`
- `int m_width;`
- `};`
- `#endif`

An extra attribute

- // Rectangle.cpp
- #include "Rectangle.h"
- //-----
- Rectangle::Rectangle ()
- {
- m_width = 0;
- m_description = "rectangle";
- }
- //-----
- void Rectangle::Input ()
- {
- Shape::Input ();
- cout << "Enter rectangle width: ";
- cin >> m_width;
- }

Both the width and the description must be initialised.

First the height is input using the Shape's Input() method, and then the extra information required by Rectangle is requested.

- `//-----`
- `void Rectangle::Draw () const`
- `{`
- `for (int row = 0; row < m_height; row++)`
- `{`
- `for (int col = 0; col < m_width * ASPECT_RATIO; col++)`
- `{`
- `cout << '*';`
- `}`
- `cout << endl;`
- `}`
- `cout << endl;`
- `}`
- `//-----`

Driver/Main/Test Program

- // Realisations.cpp
- // Version
- // 01 - Nicola Ritter first version written
- // 02 - Nicola Ritter
- // Refactored into smaller functions that will fit into
- // powerpoint.
- // 03 – smr, polymorphism is highlighted.
- //-----
- #include "Triangle.h"
- #include "Square.h"
- #include "Rectangle.h"
- #include <vector> // uses the std::vector. Change to use your Vector class.
- using namespace std;

- //-----MAIN-----
- `typedef Shape *ShapePtr;`
- `typedef vector<ShapePtr> ShapeVec; //std::vector of Shape pointers.`
- `// Change to use your own Vector class`
- `// typedef Vector<ShapePtr> ShapeVec;`
- //-----
- `// Subroutine prototypes – forward declaration`
- `void Draw (const ShapeVec &array);`
- `void Input (ShapeVec &array);`
- `char Menu ();`
- `Shape *GetShape (char ch);`

- `//-----`
- `// READ THIS TOGETHER WITH THE REALISATIONS CODE PROJECT`
- `int main()`
- `{`
- `ShapeVec array;`
- `Input (array);`
- `Draw (array);`
- `cout << endl;`
- `return 0;`
- `}`
- `//-----`

- //-----**Polymorphism in action-**-----

- **void Draw (const ShapeVec &array)**

- {

- int size = array.size();

- **for (int index = 0; index < size; index++)**

- {

- array[index]->Draw();

- }

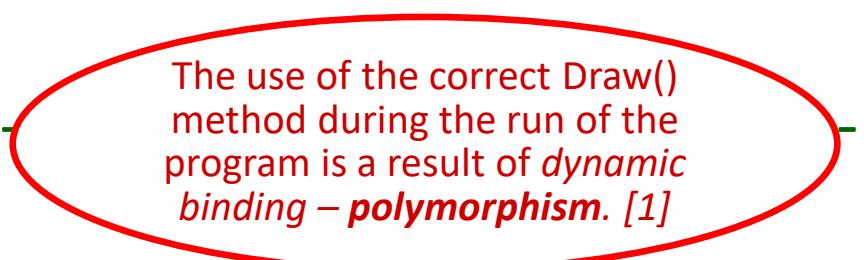
- cout << endl;

- }

- //-----



The arrow dereference symbol is used when a method is called on a pointer to an object, rather than on the object itself. [1]



The use of the correct Draw() method during the run of the program is a result of *dynamic binding – polymorphism*. [1]

```
• void Input (ShapeVec &array)
• {
•     char ch = Menu();
•     while (ch != 'Q')
•     {
•         ShapePtr shape = GetShape (ch);
•         array.push_back(shape);
•
•         ch = Menu();
•     }
•
•     for (int index = 0; index < array.size(); index++)
•     {
•         array[index]->Input(); //Polymorphic input method
•     }
•     cout << endl;
• }
```

Get a choice from the user and then get a shape based on this choice. Finally add the pointer to the shape to the array

Next get the dimensions of the shapes

- //-----
- char Menu ()
- {
- string str;
- do
- {
- cout << "S - Square" << endl;
- cout << "T - Triangle" << endl;
- cout << "R - Rectangle" << endl;
- cout << "Q - Quit entry" << endl;
- cin >> str;
- } while (strchr("STRQstrq", str[0]) == NULL); // what does this do?
- return toupper(str[0]);
- }

We input a string not a single character so that we do not have to remember to read the <enter> key.

Users are forced to input a correct value.

- //-----
- ShapePtr GetShape (char ch) // returns a pointer to parent
- {
- ShapePtr shape = NULL;
- switch (ch)
- {
- case 'S':
- shape = new Square;
- break;
- case 'T':
- shape = new Triangle;
- break;
- case 'R':
- shape = new Rectangle;
- break;
- }
- return shape;
- }

A pointer to a Shape can point to any class derived from Shape. [1]

Note that we are not breaking the rules as we are passing back a *pointer* function-wise, not an object.

Interfaces

- Occasionally an abstract class is defined where
 - there are *no* attributes defined;
 - all the methods are pure virtual methods – no body
- **Interfaces are abstractions**
 - If you write your application to abstractions, the implementation of the abstraction can change without breaking your application. [1]
- This type of class is called an *interface* and is used as just that: it defines the way in which all derived classes will interface with other parts of your software.
- In UML, they are shown with the word <<interface>> in double arrow braces above the name of the interface:



Interfaces

- It is also a good idea to name interfaces starting with the letter “I” (*IDraw* instead of *Draw*).
- The name should be in italics (*IDraw*) along with all other abstract methods.
- There is an alternative way to represent interfaces using a lollipop or circle used in component diagrams as opposed to class diagrams that we are doing. We wouldn’t use lollipop representation in this unit.

Interfaces, and Strategy design pattern [1]

- The strategy design pattern “favours composition over inheritance”.
 - *Inheritance gets abused when it is used to implement code re-use (implementation hierarch instead of type hierarchy)*
 - *Code in this case refers to algorithms implementing behaviours.*
 - *The Strategy design pattern shows how to abstract behaviours into behavioral interfaces.*
 - *Allows program clients to be written to interfaces not to implementations. (reminder: Interfaces are abstractions)*
 - *You can see the problem created by inheritance for code reuse in this video on Strategy Pattern*
<https://www.youtube.com/playlist?list=PLrhzvlci6GNjpARdnO4ueTUA VR9eMBpc> [2]
 - *The video refers to REQUIRED READING chapter 1 in the book Head first design patterns available as ebook from this unit's Myunit readings* <https://rl.talis.com/3/murdoch/items/7F748952-D8DF-E667-EA75-5603FDB25D83.html?lang=en>

Readings

- Textbook: Chapter on Classes and Data Abstractions.
- Textbook: Chapter on Inheritance and Composition, entire section on Inheritance up to but not including the short section on Composition.
- Chapter on Pointers, Classes, Virtual Functions, Abstract Classes, and Lists, start at section on Inheritance, Pointers and Virtual Functions.
- Chapter 1 of the ebook Head first Design Patterns in the Myunit readings for ict283. Available as ebook
<https://rl.talis.com/3/murdoch/items/7F748952-D8DF-E667-EA75-5603FDB25D83.html?lang=en>
 - *Or get the strategy design pattern explained to you*
<https://www.youtube.com/playlist?list=PLrhzvlci6GNjpARdnO4ueTUA VR9eMBpc>



Murdoch
UNIVERSITY

Data Structures and Abstractions

Encapsulation and Linked Lists

Lecture 17



Records using struct

- C++ records (structs): [1]

```
typedef struct
{
    string firstname;
    string surname;
    int     age;
} Person;
```

Records using class

- Encapsulating a record in a class:

```
class Person //any special behaviour for Person?  
{  
    ...  
private:  
    string firstname;  
    string surname;  
    int     age;  
};
```

When to Encapsulate?

- The question is, which should you use?
- If there are *any* input processing or output methods to be performed on a data structure *or* it is composed of other objects, then it should be encapsulated. [1]
- And, of course, if you encapsulate things in a class, then you can test all the methods and operators *in isolation* before having to combine the code with the rest of your program. **UNIT TEST [1]**

Arrays vs Lists

- We know how to declare and use “raw” array.
[1]
- We have looked at how to declare and use a list.
- The main differences are:

An array has an initial size	A list starts with 0 size
It is difficult to change the size of an array	lists automatically resize as they grow
Arrays have no inbuilt functions	lists have lots of inbuilt functions

- Obviously the list is better when it comes to memory use.

When to Encapsulate

- The rule for records also holds true for arrays and lists. As long as they are data stores that require little in the way of processing, then you can just use them as is.
- However, if you need to do bounds checking or have any processing that needs doing, *or* they contain objects, then they should almost certainly be encapsulated.
- And, of course, if you encapsulate things in a class, then you can test all the methods and operators *in isolation* before having to combine the code with the rest of your program. (as before). **ALWAYS UNIT TEST** before use in your program.

Advantages of Encapsulation

- The class can be tested in isolation before being used in a program. **UNIT TEST**
- Changes and new code can be tested in isolation before being used in a program. **UNIT TEST**
- This means that the *testing* of the program becomes modular and hence easier, and more likely to be thorough.
- Which in turn means that programs are more likely to be robust and errors are easier to find.
- It is easier to re-use code.
- Bounds checking is done in one file.
- Code is less complicated, and therefore easier to maintain.
- It becomes easy to alter *how* something is done without altering the main (client or user) program.
- It becomes easy to alter how something is stored without altering the main (client or user) program
- Memory can be more easily allocated dynamically in a safe manner.

Linked Lists

- We took a first look at linked lists in an earlier lecture note.
- Linked lists are an abstract class that model a particular type of behaviour. In a linked list, you have:
 - each node contains data and a pointer;
 - the data can only be accessed in a serial manner from the previous piece of data;
 - access to the container as a whole is done via the first element;
 - the last element must point to NULL (nullptr) to ensure algorithms cannot process past the end of the list.



The Linked List

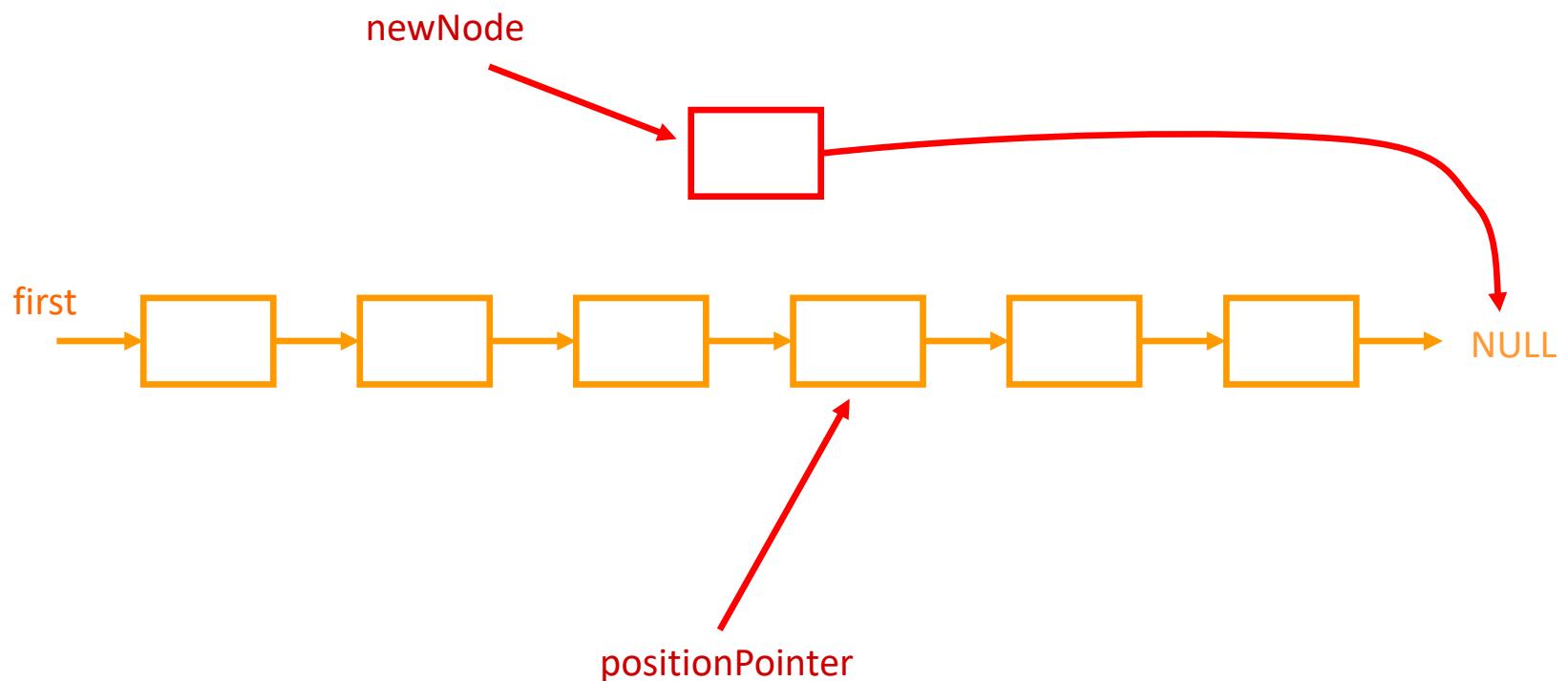
- We looked at the Node class (data plus pointer).
- A linked list class simply contains a Node or pointer to a Node.
- If it contains an actual node, it makes processing easier, but wastes the space of that Node.
- The node is called a ‘dummy header’ as it stores no actual information (data which the other nodes store).

Linked Lists vs Arrays

- Linked lists are containers as are arrays/lists.
- Unlike an array/list, you cannot access data directly in a linked list.
- Therefore access to an array element is done in constant time, but to a linked list element takes $O(n)$.
- However, if you want to insert or delete into an array it takes $O(n)$ time, whereas with a linked list it takes constant time.

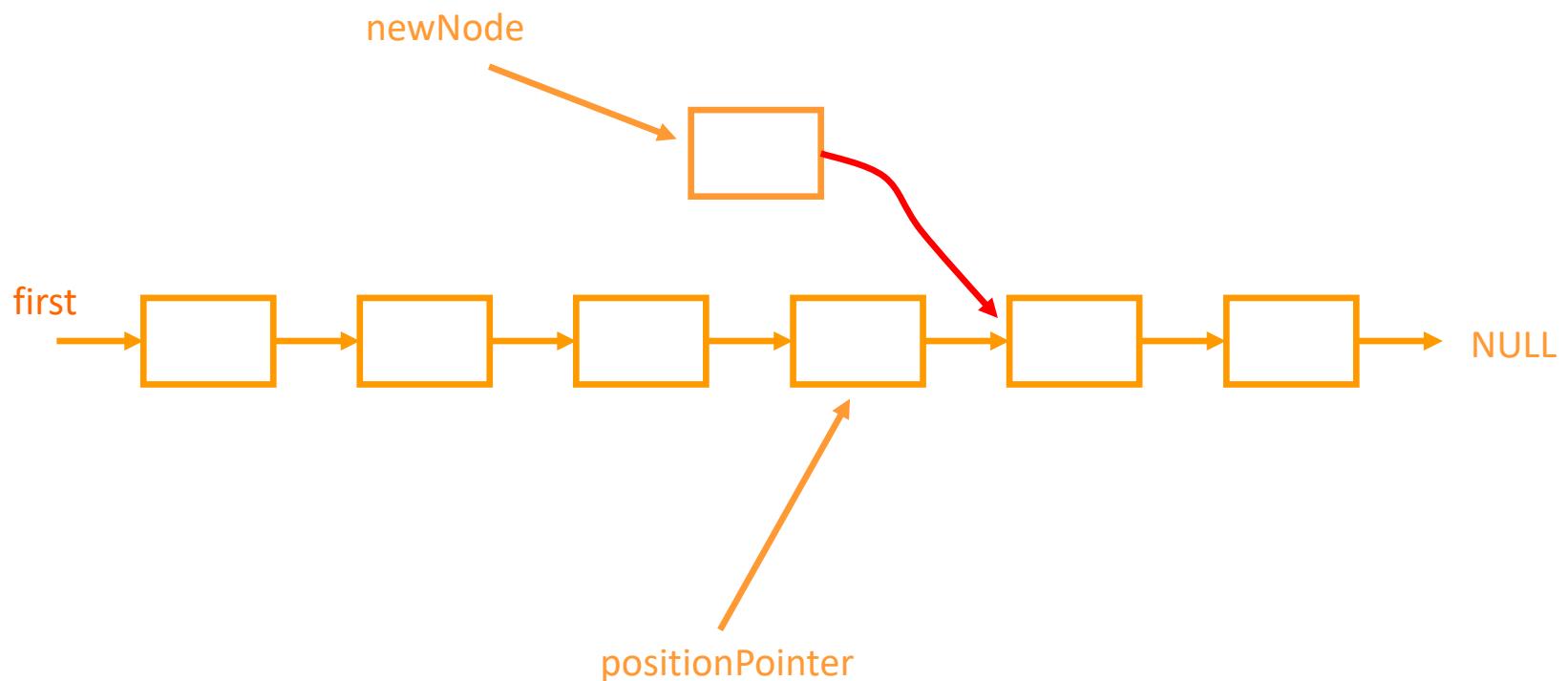
Insertion into a List [1]

- Locate node in front of the insertion point



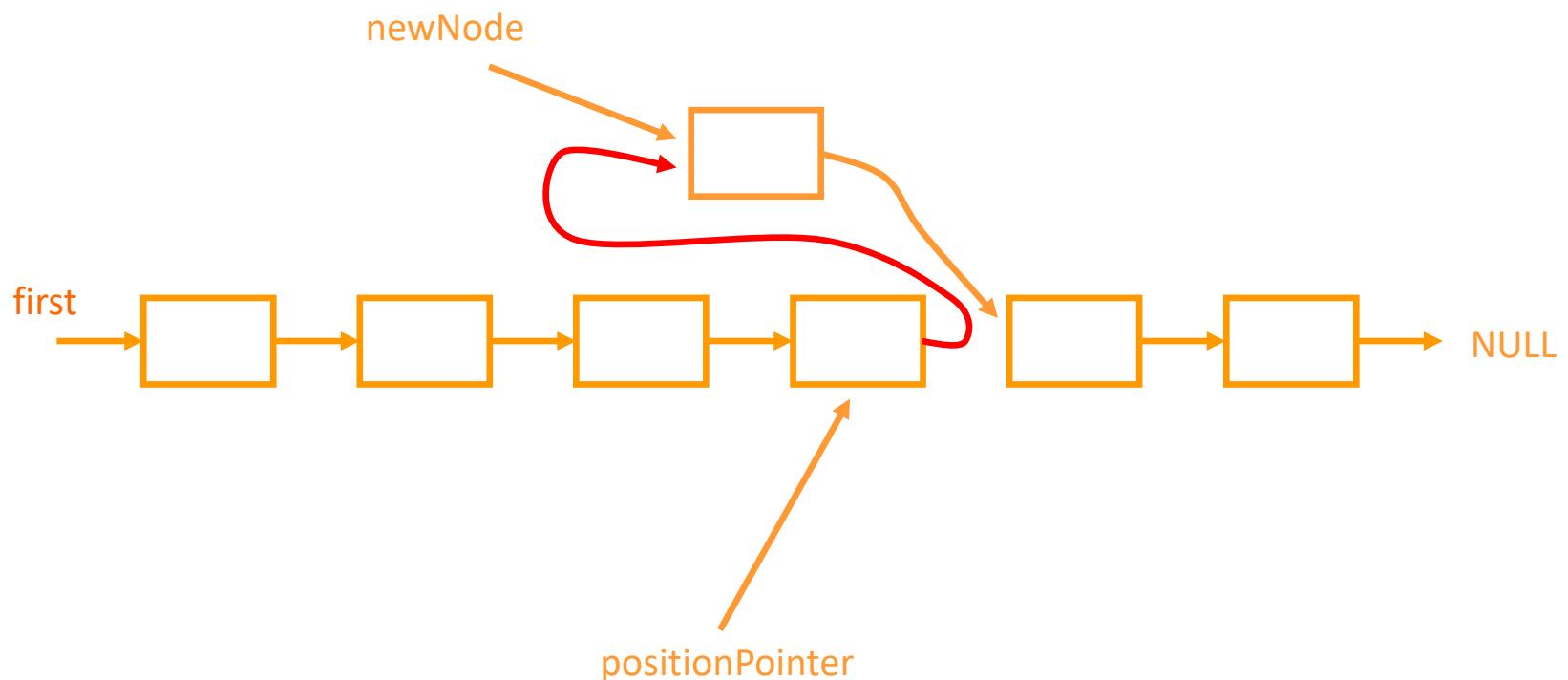
Insertion into a List

- Reassign the 'next' pointer of the new node



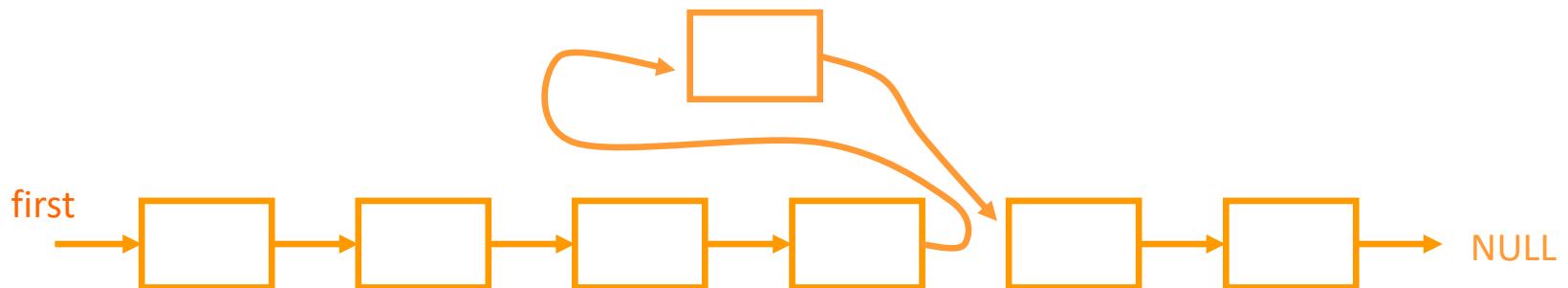
Insertion into a List

- Reassign the 'next' pointer of the node in front of the new node



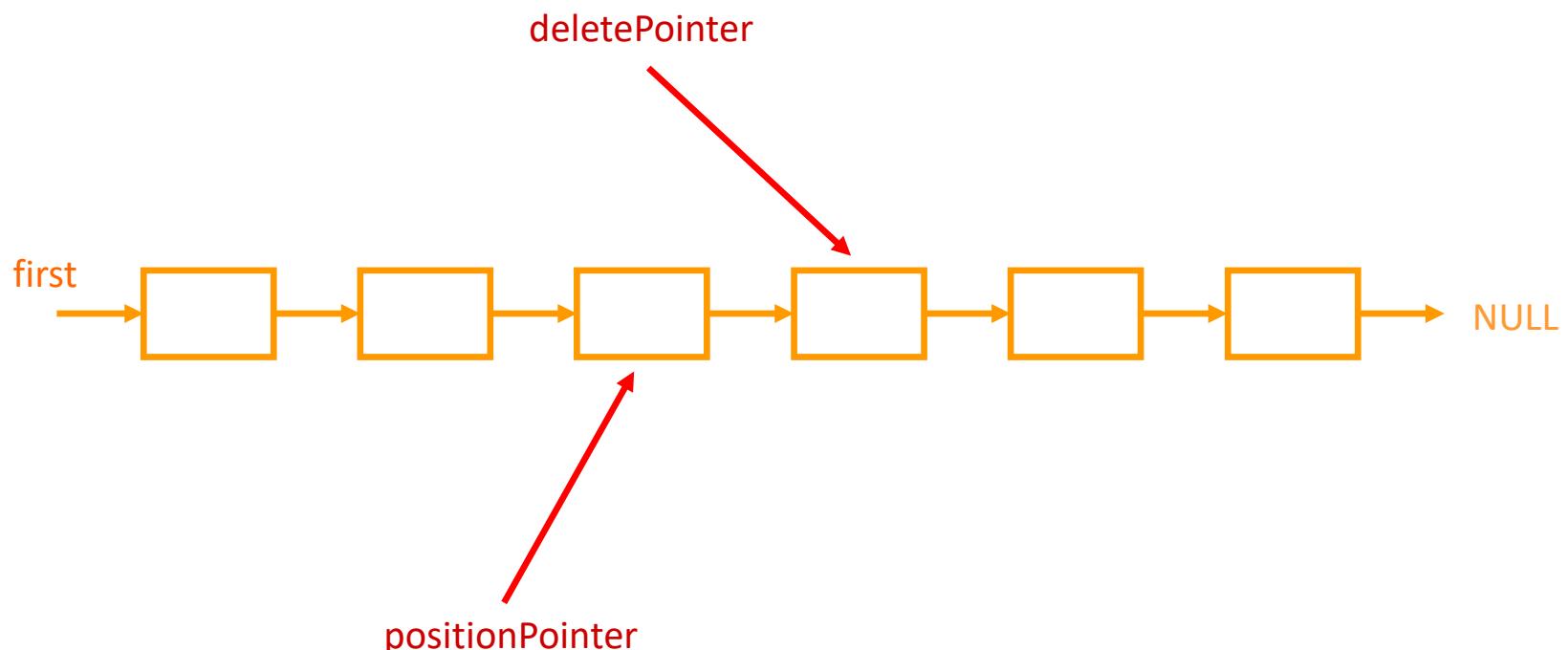
Insertion into a List

- The two other pointers are no longer needed as the node is now part of the list.



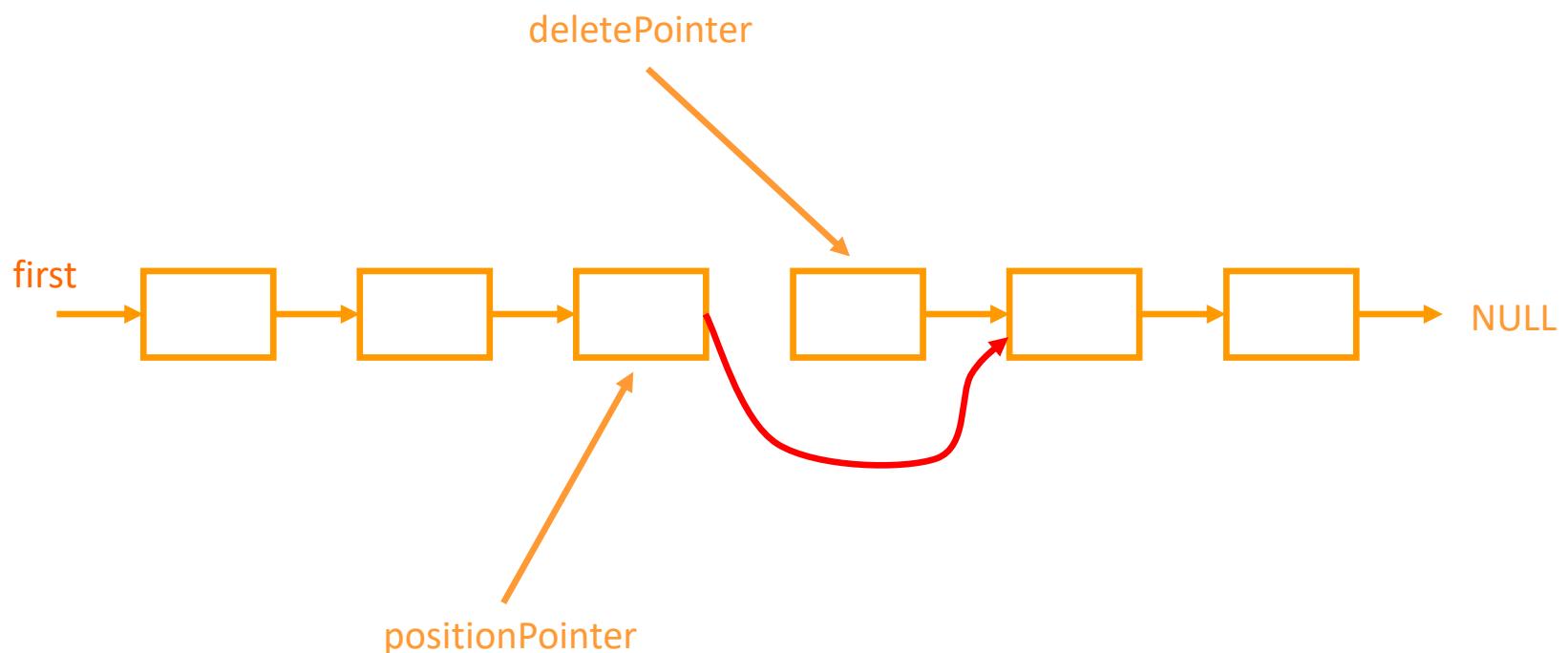
Deletion from a List

- Locate the node in front of the node to be deleted, as well as the node to be deleted.



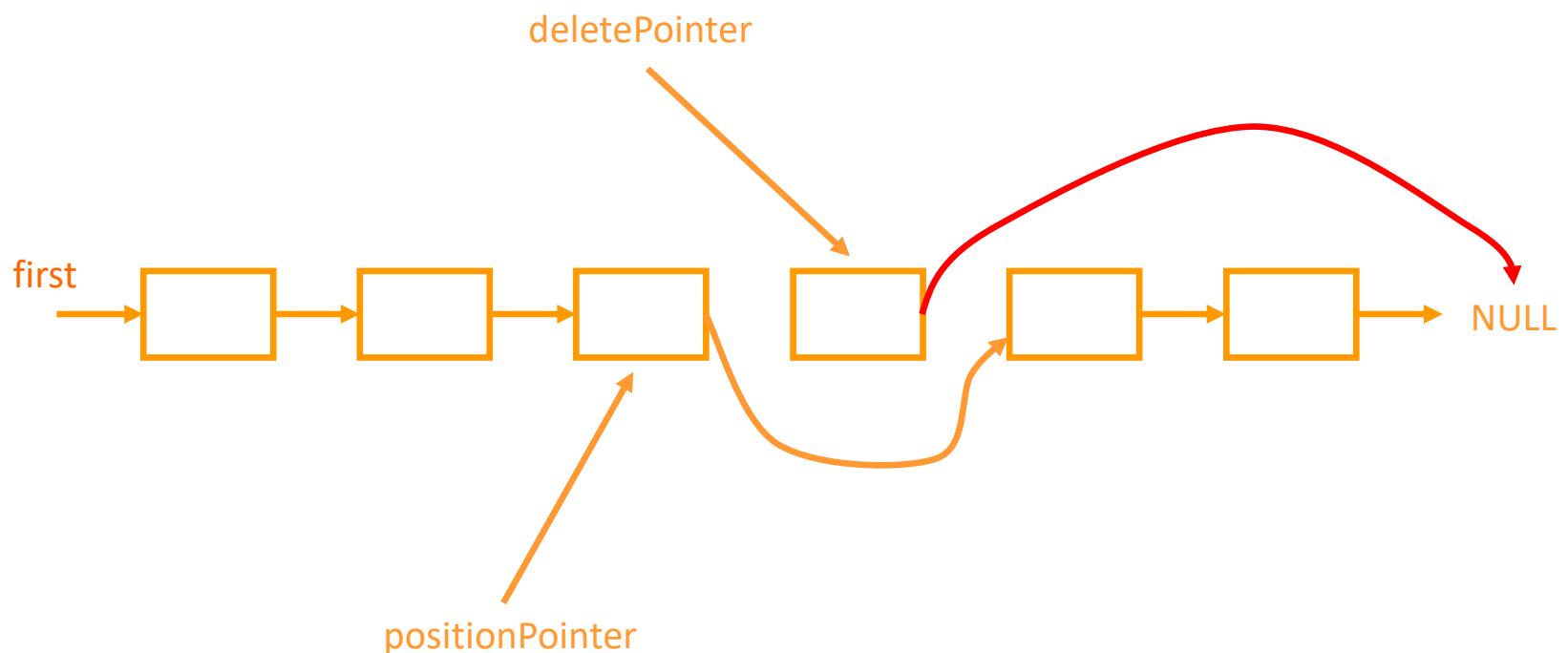
Deletion from a List

- Reassign the 'next' pointer of the node in front of that to be deleted.



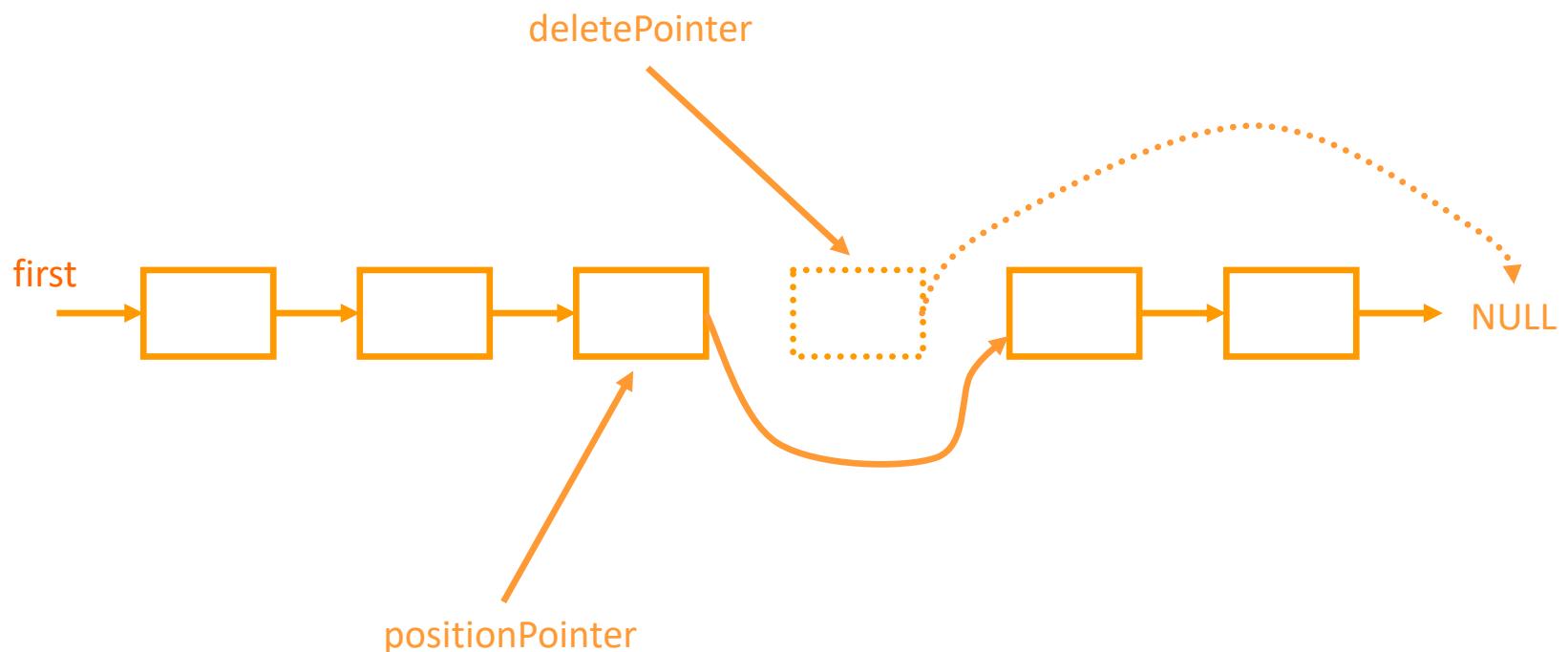
Deletion from a List

- Reassign the 'next' pointer of the node to be deleted, setting it to NULL.



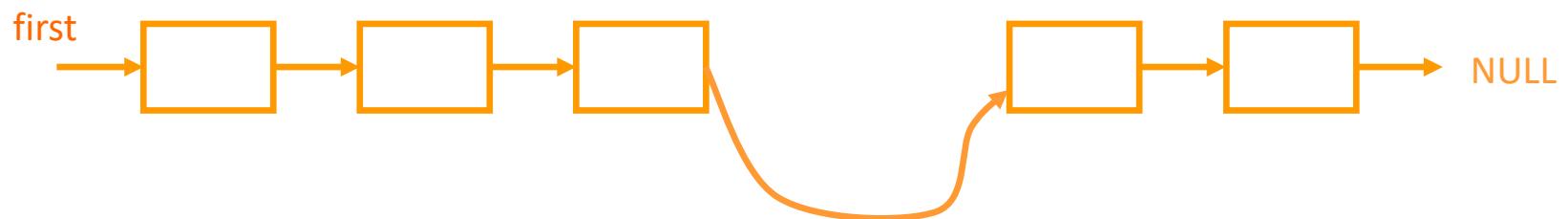
Deletion from a List

- Release the old node's storage back to the OS, using 'delete'.



Deletion from a List

- The two pointers are no longer needed as the node is no longer part of the list.



The STL List

- The STL has a linked list template.
- Just as the vector replaces the array, the list template replaces ‘home-coded’ linked lists. [1]
- As with the list, sometimes it needs to be encapsulated, sometimes it doesn’t.
- If you find yourself repeating code that accesses a linked list, then encapsulate it!
- The list requires the `<list>` header file.
- It is declared in the same way as a list:

```
typedef list<int> IntList;  
IntList mylist;
```

STL list Methods

<code>mylist.clear ()</code>	Empties the list.
<code>mylist.empty ()</code>	Returns true if the list is empty.
<code>mylist.erase (<various>)</code>	Erases a part of the list.
<code>mylist.insert (<various>)</code>	Add data to the list.
<code>mylist.push_back (data)</code>	Add one piece of data to the end of the list.
<code>mylist.pop_back ()</code>	Delete the last item in the list.
<code>mylist.push_front (data)</code>	Add one piece of data to the front of the list.
<code>mylist.pop_front ()</code>	Delete the first item in the list.
<code>mylist.begin()</code>	Returns an iterator that points to the first item in the list.
<code>mylist.end()</code>	Returns an iterator that points to just after the last item in the list.
<code>mylist.size()</code>	Returns the size of the list.
<code>mylist.sort()</code>	Sorts the list.
<code>mylist.swap (mylist2)</code>	Swaps the contents of the two lists.

Seem Familiar?

- Yes, these are almost exactly the same methods as listed for the STL std::vector class.
- The huge advantage of the STL is that the classes all have almost identical methods and operators.
- There are a few that are unique to one or other class, but on the whole they are the same.
- Here, the two that are in list and not in vector are push_front, pop_front and sort.
- Almost all of the STL classes can also all be passed to the same algorithms in the algorithm class.
- If they can't then the compiler will soon let you know!

Advantages of Encapsulation Again

- Let's suppose you want a container of Lights.
- When you first code it you use a vector of Lights as the data structure.
- After a while you realise that a linked list would be a better container and you decide to change to a list.
- If you had **not encapsulated it**, you now must go through *possibly* thousands of lines of **code in multiple files to alter it from a vector to a list.**
- If you encapsulated it, you probably only have to change a few lines in only two files. This is because the underlying container would have been private and all the other code in the various files would not have direct access to it.
 - If you designed your encapsulation well, there would be no need to change the public access methods just because the underlying container was changed from a vector to a list.
- A very big-time saver!!

Readings

- Textbook: Chapter on Linked Lists.
 - Go through the programming Example on video store at the end of the chapter.
- Chapter on Standard Template Library
- Re-read chapter 1 “The Object Oriented paradigm in Design Patterns Explained: A New Perspective on Object-Oriented Design. See Topic 1 readings. Available as an ebook from the library.

Further exploration

- For a more details of linked lists with some level of language independence, see the reference book, Introduction to Algorithms section on “Linked Lists” in the chapter on “Elementary Data Structures” (10).
- For more on STL containers see
<http://www.cplusplus.com/reference/stl/>



Murdoch
UNIVERSITY

Data Structures and Abstractions

Two Dimensional Structures

Lecture 18



Two Dimensions

- Two dimensional structures are complicated.
- Therefore they should always be encapsulated.
- This also gives great freedom in how they should be implemented.
- And great freedom to change the implementation if required.
- Some possibilities are:
 - an old-fashioned two dimensional array
 - an array of vectors
 - a vector of arrays
 - an array of lists
 - etc
 - in other words an array/list/vector of array/list/vector
 - limited only by human imagination, as multidimensions are possible

Which One?

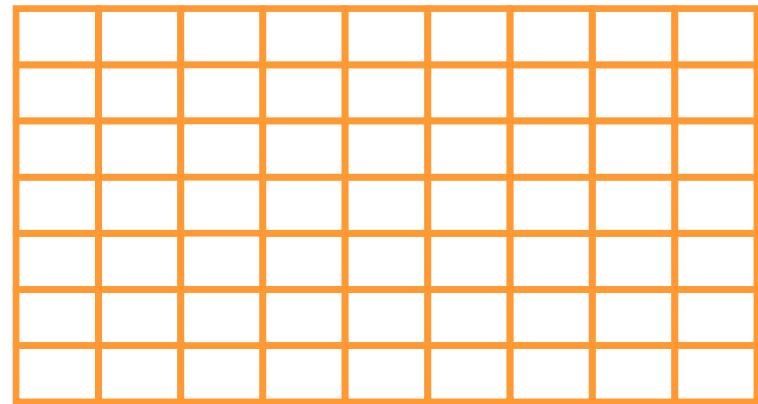
- The choice will depend on the what you are trying to model.
- Ask yourself:
 - Do you know the dimensions in advance?
 - Are there always going to be the same number of columns in each row?
 - Does there need to be a set number of rows, even if there is nothing in each row?
 - Will you need to add/delete rows or columns at the ends?
 - Will you need to insert/delete rows or columns in the middle?
 - Will you need to insert/delete a single piece of data at the end of a single row?
 - Will you need to insert/delete a single piece of data in the middle moving along the other data in that row only?
 - Do you need direct access to the data?
- When you can answer all these questions, you will be able to choose the correct combination of data structures for the task.

An Old Fashioned 2D Array

- `// A two dimensional array of DataType objects`
- `typedef DataType TableType[ROWS][COLS]; // [1]`

- `...`

```
• class Table
  {
  • public:
  •     ...
  • private:
  •     TableType m_array;
  }
```



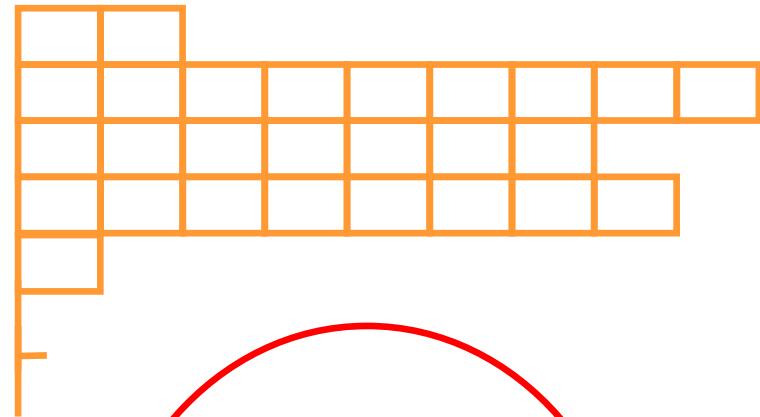
Uses exactly ROWS x COLS slots of the size of DataType

Possible Application
Icon Storage, but can be anything else

An Array of Vectors

- `// A vector of DataType objects`
- `typedef vector<DataType> Row;`
- `// Rows of these vectors`
- `typedef Row TableType[ROWS];`
- `...`
- `class Table`
- `{`
- `public:`
- `...`
- `private:`
- `TableType m_array;`
- `}`

Possible Application
Accumulator for a fixed rows and
variable columns



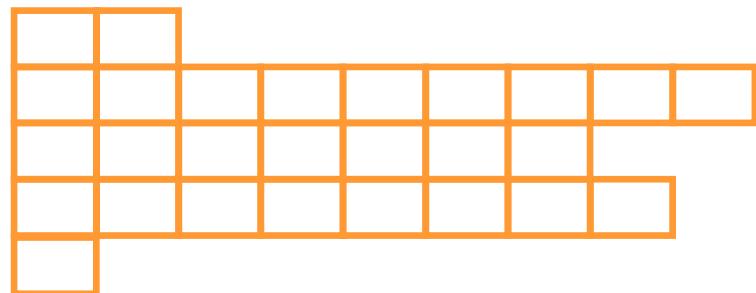
Each row can be a different size, but there are still exactly ROWS number of rows, even if some are empty.
The STL vector takes care of the rows.

A Vector of Vectors

- // A vector of DataType objects
- **typedef** **vector<DataType>** Row;
- // vector of these vectors
- **typedef** **vector<Row>** TableType;

• ...

```
• class Table
  • {
  •   public: Possible Application
  •          Accumulator for an unknown number
  •          of items
  • ...
  • private:
  •   TableType m_array;
  • }
```

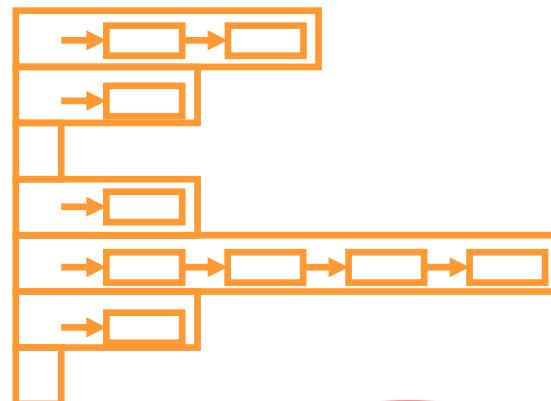


Each row can be a different size, and now we only have the rows we actually want. Variable columns

An Array of Lists

- `// A list of DataType`
 - `typedef list<DataType> DTlist;`

 - `// An array of these lists`
 - `typedef DTList TableType[ROWS];`



Possible Application

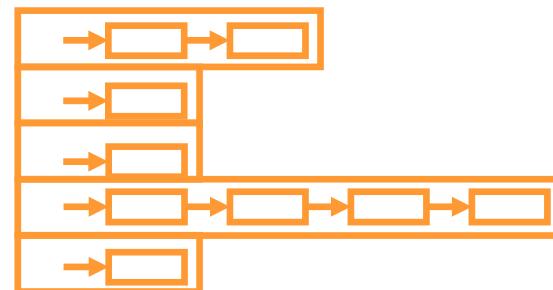
Each list initialises itself, so this structure is safer than the last few. There are ROWS number of lists, which may or may not be the best structure.

A Vector of Lists

- `// A list of DataType`
- `typedef list<DataType> DTlist;`
- `// A vector of these lists`
- `typedef vector<DTlist> TableType;`

• ...

```
• class Table
  • {
  • public:
  •     Possible Application
  •     Lists of students grouped under
  •     country of origin
  • ...
  • private:
  •     TableType m_array;
  • }
```



Once again we
only have the
number of rows
required. Grow
as needed

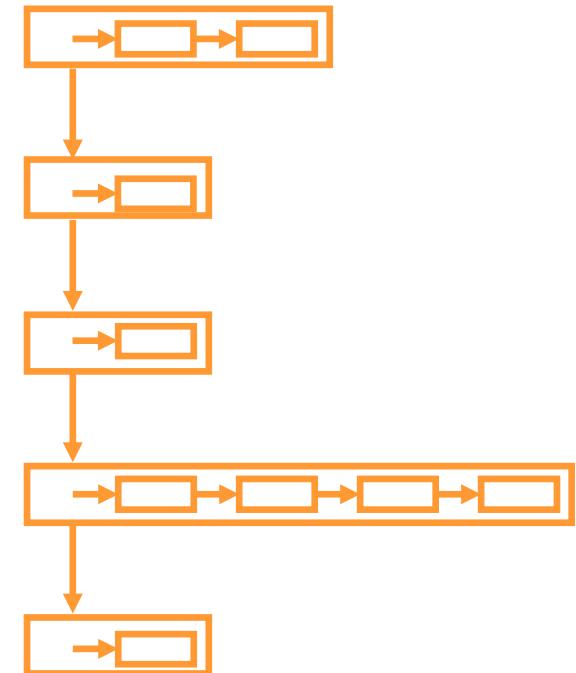
A List of Lists

- // A list of DataType
- **typedef** list<DataType> DTlist;
- // A list of these lists
- **typedef** list<DTlist> TableType;

• ...

```
class Table
{
public:
...
private:
    TableType m_array;
}
```

Possible Application
A list of people on the carriages of a train.



Complicated but
versatile!

A List of Arrays

- `// A list of DataType`
 - `typedef DataType Array2D[ROWS][COLS];`
 - `// A list of these lists`
 - `typedef list<Array2D> TableType;`

...

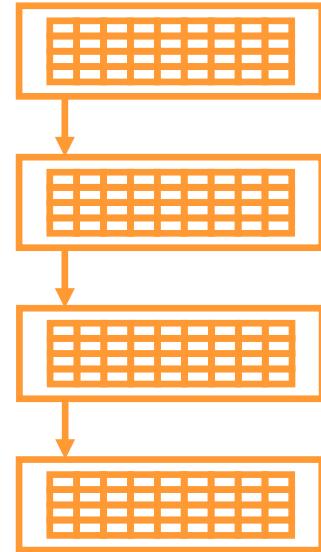
```
class Table
{
public:
    ...
private:
    TableType m_array;
}
```

Possible Application
A simulation of the seats in the carriages of a train.

This
consider
three
struct
should
list
Ca

Possible Application

A simulation of the seats in the carriages of a train.



This can be considered to be a three dimensional structure, and should be coded as a list of (e.g.) Carriages.

Etcetera!

- The possibilities are 2^n , where n is the number of different types of 1D structure.
- Choosing the right one is the only difficulty.
- But if you encapsulate it, changing your mind only costs some time and effort within 2 files the interface (header) and implementation (source) files.
- Change when not encapsulated could mean a great deal of work indeed!

Full Encapsulation

- Layering the encapsulation makes maintenance easier and easier.
- Therefore rather than making the inner layer the raw container type, it would be a class.
- As would the outer layer.
- As well as making maintenance easier, it will make processing simpler and clearer.
- And, of course, the structure is clearer.

Readings

- Textbook: Chapter on Arrays and Strings, sections on Parallel Arrays, Two and Multidimensional Arrays.



Murdoch
UNIVERSITY

Data Structures and Abstractions

Sets

Lecture 19



Sets

- By definition, Sets are unordered collections of data.
 - $A = \{2, 1\}$
 - $B = \{1, 2, 2, 1, 8/4\}$
 - $C = \{x: x^2 - 3x + 2 = 0\}$
 - Note that $A = B = C$ i.e., the sets above are equal to each other [1]
- They are used in maths as well as in many other fields.
- But in the computing domain, there are some variations in the way sets are dealt with.
 - Some sets can contain the actual data values, whilst others only keep a record of the presence or absence of data values. STL Bitsets
 - Elements of a set may not be repeated – a common variation [2]. If repeated elements are needed, then a multiset or bag is used. STL multiset
 - A set is explicitly defined to be unordered, but some implementations require ordering for efficiency reasons [3]. The STL set is an associative container and in STL associative containers are ordered.
 - The last two variations break the Set abstraction, but STL designers decided that is fine so sets and multisets are provided. [3]

Sets

- There are some unique operations for sets:
 - subset
 - union
 - intersection
 - difference
 - element

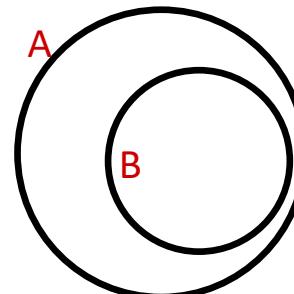
Subset \subset

- **Set B is a subset of Set A if all elements in B are also elements of A.**

- **Example 1:**

if $A = \{a, b, c, d, g\}$ and $B = \{c, g\}$

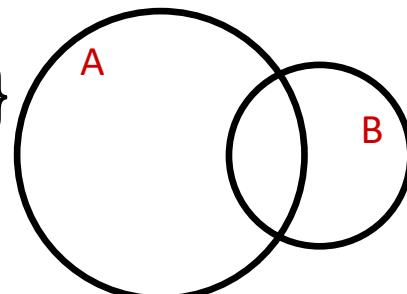
then B is a subset of A.



- **Example 2:**

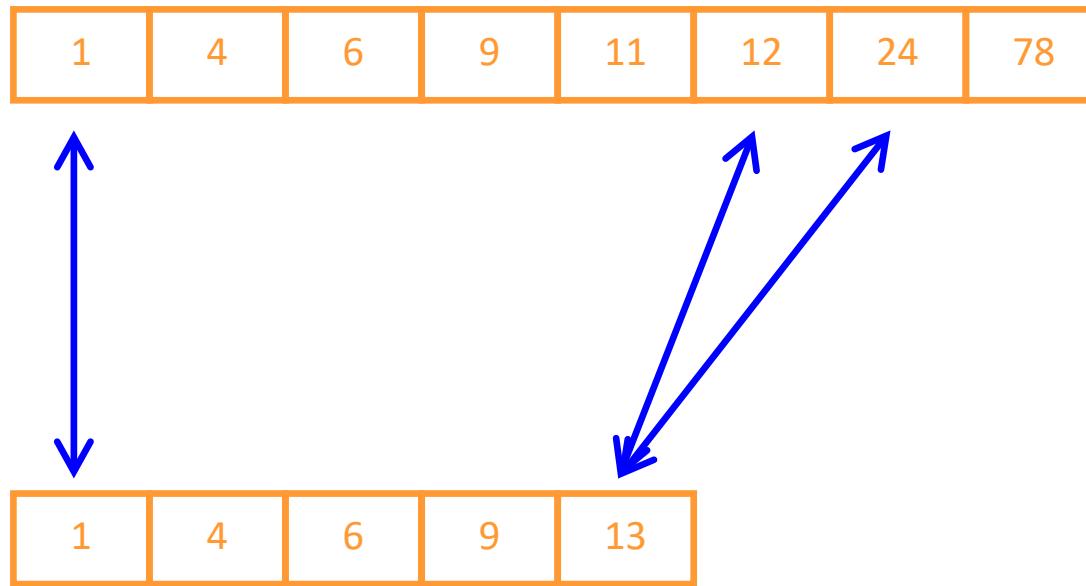
if $A = \{a, b, c, d, g\}$ and $B = \{c, g, u, w\}$

then B is not a subset of A.



Subset Animation [1]

subset = **false**



Subset Pseudo-code

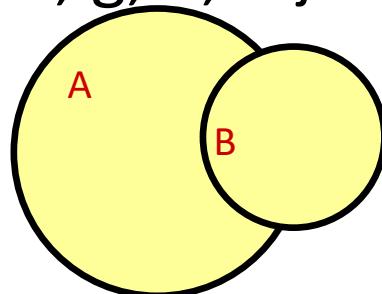
```
IsSubsetOf (other) [1]
```

```
    Boolean subset = true
    WHILE more elements in this set AND
        more elements in the other set AND
        subset = true
        IF this element = other element
            Get next element from each set
        ELSE IF this element < other element
            subset = false
        ELSE
            Get next element from other set
        ENDIF
    ENDWHILE
    IF more elements in this set
        subset = false
    ENDIF

END IsSubsetOf
```

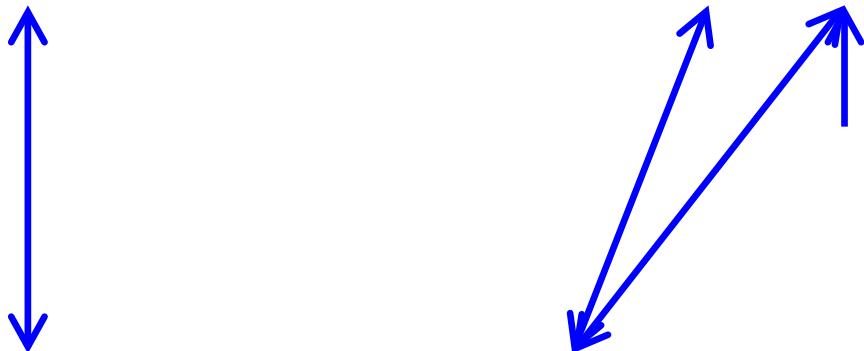
Union (or, ||)

- The union of Set A and Set B is the collection containing all elements that are in *either* of them, removing double ups. [1]
- For example:
if $A = \{a, b, c, d, g\}$ and $B = \{c, g, u, w\}$
then $C = A \text{ or } B = \{a, b, c, d, g, u, w\}$
- C is shown in yellow:



Union Animation

1	4	6	9	11	12	24	78
---	---	---	---	----	----	----	----



newSet

1	4	6	9	11	12	13	24	78
---	---	---	---	----	----	----	----	----

Union Pseudo-code

```
Union (other, newSet) [1]
```

```
    WHILE more elements in this set AND
        more elements in the other set
        IF this element = other element
            Add this element into newSet
            Get next element from each set
        ELSE IF this element < other element
            Add this element to newSet
            Get next element from this set
        ELSE
            Add other element to newSet
            Get next element from other set
        ENDIF
    ENDWHILE
```

```
    WHILE more elements in this set
        Add this element to newSet
        Get next element from this set
    ENDWHILE
```

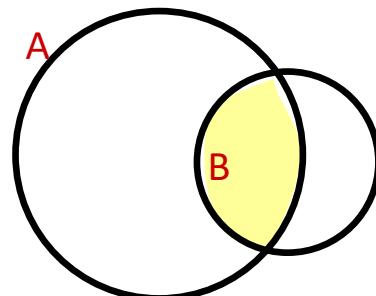
```
    WHILE more elements in other set
        Add other element to newSet
        Get next element from other set
    ENDWHILE
```

```
END Union [2]
```

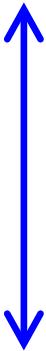
Intersection (and, &&)

- The intersection of Set A and Set B is the collection containing all elements that appear in *both* of them.
- For example:

if $A = \{a, b, c, d, g\}$ and $B = \{c, g, u, w\}$
then $C = A \text{ and } B = \{c, g\}$
- C is shown in yellow:



Intersection Animation



newSet



Intersection Pseudo-code

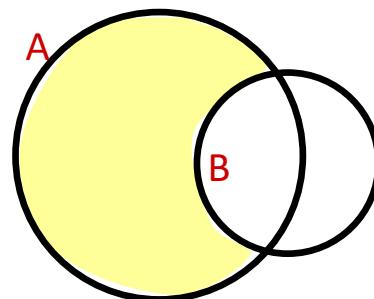
```
Intersection (other, newSet) [1]

WHILE more elements in this set AND
      more elements in the other set
      IF this element = other element
          Add this element into newSet
          Get next element from each set
      ELSE IF this element < other element
          Get next element from this set
      ELSE
          Get next element from other set
      ENDIF
ENDWHILE

END Intersection
```

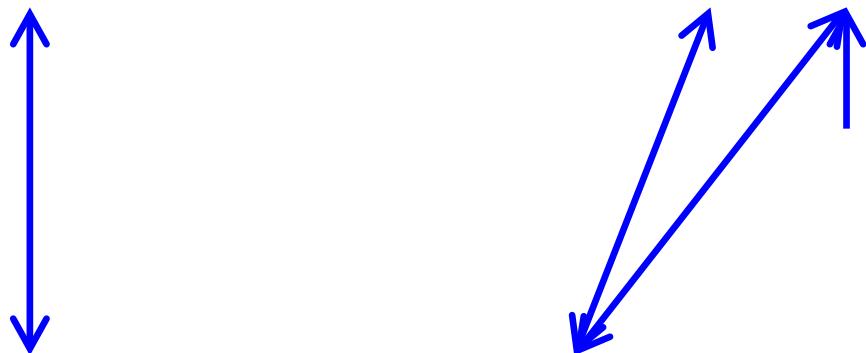
Difference (-)

- The difference of Set B from Set A is the collection containing all elements that are in A but not in B.
- For example:
if $A = \{a, b, c, d, g\}$ and $B = \{c, g, u, w\}$
then $C = A - B = \{a, b, d\}$
- C is shown in yellow:



Difference Animation

1	4	6	9	11	12	24	78
---	---	---	---	----	----	----	----



1	4	6	9	13
---	---	---	---	----

newSet

11	12	24	78
----	----	----	----

Difference Pseudo-code

```
Difference(other, newSet) [1]

    WHILE more elements in this set AND
        more elements in the other set
        IF this element = other element
            Get next element from each set
        ELSE IF this element < other element [2]
            Add this element to newSet
            Get next element from this set
        ELSE
            Get next element from other set
        ENDIF
    ENDWHILE
    WHILE more elements in this set
        Add this element to newSet
        Get next element from this container
    ENDWHILE

END Difference
```

The STL Set

- There is an STL set in C++.
- It requires the **<set>** header file.
- As for the others it is declared using:

```
typedef set<int> IntSet;  
IntSet aset;
```

- The best place to go for information is (again as before):

<http://www.cppreference.com/cppset/index.html>

STL Set Methods [1]

<code>aset.clear()</code>	Empties the set
<code>aset.empty()</code>	Returns true if the set is empty
<code>aset.begin()</code>	Returns an iterator to the first element in the set
<code>aset.end()</code>	Returns an iterator to past the end of the set
<code>aset.erase()</code>	Erase elements from the set
<code>aset.find()</code>	Find elements in the set
<code>aset.insert()</code>	Insert elements into the set
<code>aset.size()</code>	Returns the size of the set
<code>aset.swap()</code>	Swaps the contents of two sets

Set Algorithms

- For reasons of general utility, routines that could have been placed in the STL set class were placed in the algorithm class:

```
set_difference(set1.begin(), set1.end(),
                set2.begin(), set2.end());
set_intersection(set1.begin(), set1.end(),
                 set2.begin(), set2.end());
set_union(set1.begin(), set1.end(),
          set2.begin(), set2.end());
```
- These should have been *operations* on sets, because it would have been intuitive.
 - Or (preferred) as helper functions available when #include <set> (What is the Open-closed principle?) [1]
- As these routines have general utility, they can be applied to other linear data structures like vectors. This approach can be argued to be good, as re-use of code is happening.
- And there is no subset but a subset set helper function can be written using algorithm's `includes` or `set::find()` function.

```
// Do the set difference using the insert iterator
set_difference(set1.begin(), set1.end(),
               set2.begin(), set2.end(), resultItr);
```

- An abstract representation could have operator-(..) defined, so that:

```
resultSet = set1 - set2
```

- For this reason—unless the task is trivial—the STL set needs to be encapsulated or a helper operator/function is provided.

- Prefer the helper operator/function as this means the least amount to code for a given functionality.
 - The helper operator or function uses only the set's public interface.

Readings

- Textbook: Standard Template Library, section on Associative containers relating to set and multiset.
- Library EReserve: Preiss, Data structures and algorithms with object-oriented design, Chapter 12
- <http://www.cplusplus.com/reference/stl/>
- <http://en.cppreference.com/w/cpp/container/set>
- [http://en.cppreference.com/w/Main Page](http://en.cppreference.com/w/Main_Page)



Murdoch
UNIVERSITY

Data Structures and Abstractions

MAPS

Lecture 20



Note 1 (legacy code only)

- When you compile some STL code in VC++ you might get a warning: [1]

```
ICT283\Code\Sets\SetDifference.cpp(59) : warning C4786:  
'std::pair<std::Tree<int,int, std::set<int, std::less<int>, std::allocator<int>>:: _Kfn, std::less<int>, std::allocator<int>>::const_iterator, std::Tree<int,int, std::set<int, std::less<int>, std::allocator<int>>:: _Kfn, std::less<int>, std::allocator<int>>::const_iterator>' : identifier was truncated to '255' characters in  
the debug information
```

- This is the *only* warning you can ignore completely (a debug identifier)
- If it really annoys you, then add the following code before the includes in the file that is generating the warning:
#pragma warning (disable : 4786)
- Do **not** disable any other warning!!
 - DO NOT JUST DISABLE THE WARNING IF YOU ARE NOT GETTING THE WARNING.
 - Warning is only on older implementations of Visual C++, so you are not likely to see it now. If you do see it, please let me know. As we
 - Legacy code and compilers may generate this issue, so just for noting**
 - You wouldn't see this error in the work you are doing in this unit but be aware of issues like this with legacy code and older compilers.**

Note 2 (relevant now)

- If you get an error message such as:

```
ICT283\Code\Map\Map.cpp(57) : error C2440: 'initializing' : cannot convert from
'class std::Tree<class std::basic_string<char,struct
std::char_traits<char>,class std::allocator<char> >,struct std::pair<class
std::basic.....'
No constructor could take the source type, or constructor overload resolution
was ambiguous.
```

- Then it almost always means that you are passing an object as a const reference to a function that uses iterators. Iterators expect references not const references. So, for example, the code below would probably generate this error: [1]

```
void DoSomething (const IntSet &aset) //IntSet is some type with an iterator
// typically, this type has had a typedef
// typedef set<int> IntSet;
{
    IntSet::iterator itr = aset.begin(); // itr can be used to modify - error
}
```

- To solve it, use a `const_iterator`, instead of an iterator: [2]

```
void DoSomething (const IntSet &aset)
{
    IntSet::const_iterator itr = aset.begin();
```



Maps

- An association (pairing) is a connection between two things, for example the word “sanity” (*key*) is associated with the definition (*value*) “the state of having a normal healthy mind”*
- A dictionary or *map* is then a collection of key-value associations [1].
- The first part of the pair is often called a *key*.
- The data in maps is inserted, deleted and found using the key. So key needs to be unique but value need not be. [2]
- For example, if one had a map that *was* an English dictionary, then we would expect to be able to retrieve the definition of sanity using something like:

```
dictionary.GetDefinition ("sanity");
```

or even

```
dictionary["sanity"];
```

* Australian Dictionary, Collins, 2005

The STL Map

- The STL map is a very nice template indeed.
- The declaration requires two data types, the first being the key and the second being the data to be stored in association with the key. [1]
- For example, consider a class taking a vote on who should be the class president. We want to associate names with an integer number of votes:

```
#include <map>
...
map<string, int> Popularity;
...
Popularity pop;
```

A Simple Map Program

- // Normal comments up here
- #include <map>
- #include <iostream>
- #include <iomanip>
- #include <string>
- using namespace std; **// don't do this – use the approach in the code that is provided separately.**
- //-----
- const string END = "end"; // string object
- //-----
- **typedef map<string,int> Popularity;**
- **typedef Popularity::iterator PopItr;**
- **typedef Popularity::const_iterator PopCIttr;** // see textbook chapter on STL

It can be really useful to define an iterator for each STL type you use



- `//-----`
- `void AddData (Popularity &pop);`
- `void Output (const Popularity &pop);`
- `//-----`
- `int main ()`
- `{`
- `Popularity pop;`
- `AddData (pop);`
- `Output (pop);`
- `cout << endl;`
- `return 0;`
- `}`
- `//-----`

- void AddData (Popularity &pop)
- {
- string name;
- // Prime the while loop
- cout << "Enter vote name, or \"END\" to finish: ";
- getline (cin, name);
- while (name != "end") // is this comparison efficient? [1]
- {
- // If they are part of the map already, this adds 1
- // to their score. If they are not, it puts them
- // in the map and gives them a score of 1.
- // see missing code in the notes section [2]
- cout << "Enter vote name, or 'end' to finish: ";
- getline (cin, name);
- }
- }

- //-----
-
- **void** Output (**const** Popularity &pop)
- {
- PopCltr winner = pop.begin(); // set a temp winner as the first item
-
- // For each entry in the map
- **for** (PopCltr itr = pop.begin(); itr != pop.end(); itr++)
- {
- // Output the first and second parts of the pair (association)
- cout << setw(20) << itr->first << ":" << itr->second << endl;
-
- // Now check if this person should be the winner
- **if** (winner->second < itr->second) // compare the value [1]
- {
- winner = itr;
- }
- }
-
- // Output the winner
- cout << endl << "The new class president is " << winner->first
- << " with " << winner->second << " votes" << endl; [2]
- }
-
- //-----

Readings

- Textbook: Chapter on Standard Template Library.
- Map: <https://en.cppreference.com/w/cpp/container/map>
- Multimap: <https://en.cppreference.com/w/cpp/container/multimap>



Murdoch
UNIVERSITY

Data Structures and Abstractions

Stacks

Lecture 21



Temporary Storage

- When processing it is often necessary to put data into temporary storage.
- This can happen, for example, when:
 - processing events in an event-driven OS;
 - processing email in and out of a server;
 - scheduling jobs on a main-frame;
 - doing calculations;
 - sorting or merging;
- The most common data structures for temporary storage are **stacks**, **queues**, **heaps** and **priority queues**.

Stacks

- Stacks are ADS that emulate, for example, a stack of books: you can only put things on or take them off at the top.
- There are only two operations allowed on a stack: **[1]**
 - **Push (something on to it)**
 - **Pop (something off it)**
- Plus two query methods:
 - **Empty ()**
 - **Full () // optional**
- Since the last thing on is the first thing off, they are known as LIFO (Last In, First Out) data structures, or sometimes FILO (First In, Last Out).
- In essence, a stack reverses the order of the data.

Stack Implementation

- Stacks 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 **Push**, **Pop**, **Empty** and (optionally) **Full**, then it is a stack.
- Most commonly they are implemented using arrays, lists or an STL structure.
- If none of these exactly fit the required abstraction that we are after, they should be encapsulated inside our own Stack. [1]

Error Conditions for Stacks

- If you try to **Push ()** onto a stack that has no free memory, then you get overflow.
- If you try to **Pop ()** from an empty stack then you have underflow.
- So **Push ()** and **Pop ()** return a boolean to indicate if one of these errors has occurred.

Stack Example (Animation)

Array Implementation



`m_top`

Linked List Implementation



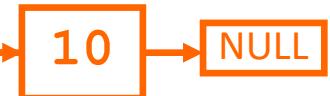
Stack Example (Animation)

Push (10)

Array Implementation



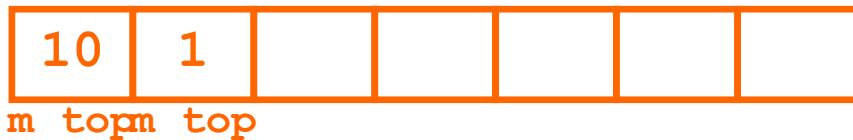
Linked List Implementation



Stack Example (Animation)

Push (1)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (23)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Pop (num)

num 23

Array Implementation



Linked List Implementation

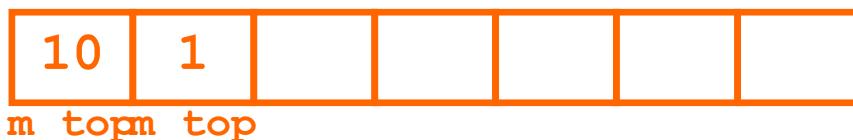


Stack Example (Animation)

Pop (num)

num 1

Array Implementation



Linked List Implementation



Stack Example (Animation)

Pop (num)

num 10

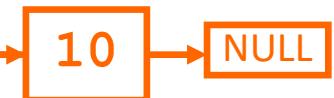
Array Implementation



10

`m_top`

Linked List Implementation



10

NULL

Stack Example (Animation)

Pop (num)

num

Array Implementation



m_top

Linked List Implementation



m_top

Stack Example (Animation)

Push (12)

Array Implementation



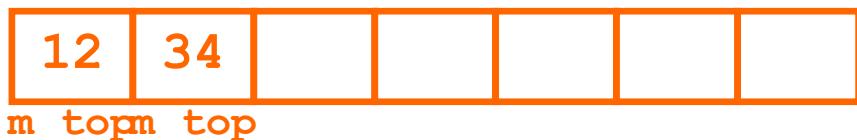
Linked List Implementation



Stack Example (Animation)

Push (34)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (23)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (36)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (98)

Array Implementation



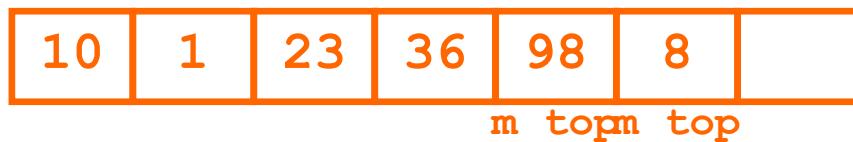
Linked List Implementation



Stack Example (Animation)

Push (8)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (76)

Array Implementation



Linked List Implementation



Stack Example (Animation)

Push (66)

Array Implementation



Linked List Implementation



Array Push Algorithm

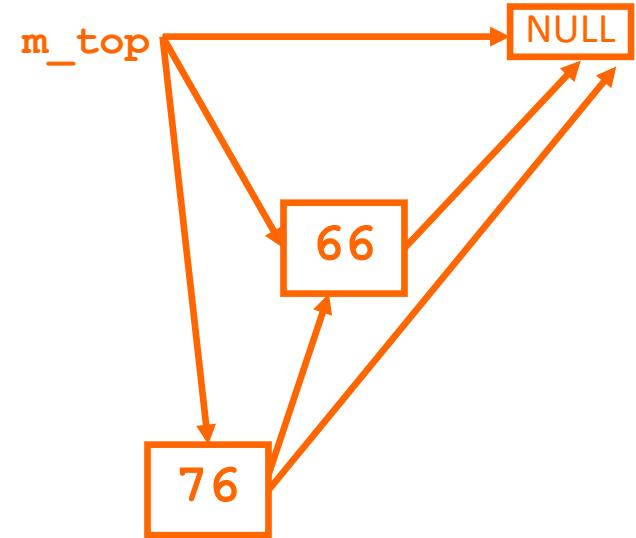
- PUSH (DataType data): boolean
- IF $m_top \geq \text{ARRAY_SIZE}-1$
- return FALSE
- ELSE
- Increment m_top
- Place data at position m_top
- return TRUE
- ENDIF
- END Push

Array Pop Algorithm

- POP (DataType data): boolean
- IF m_top < 0
- return FALSE
- ELSE
- data = data at position m_top
- Decrement m_top
- return TRUE
- ENDIF
- END Pop

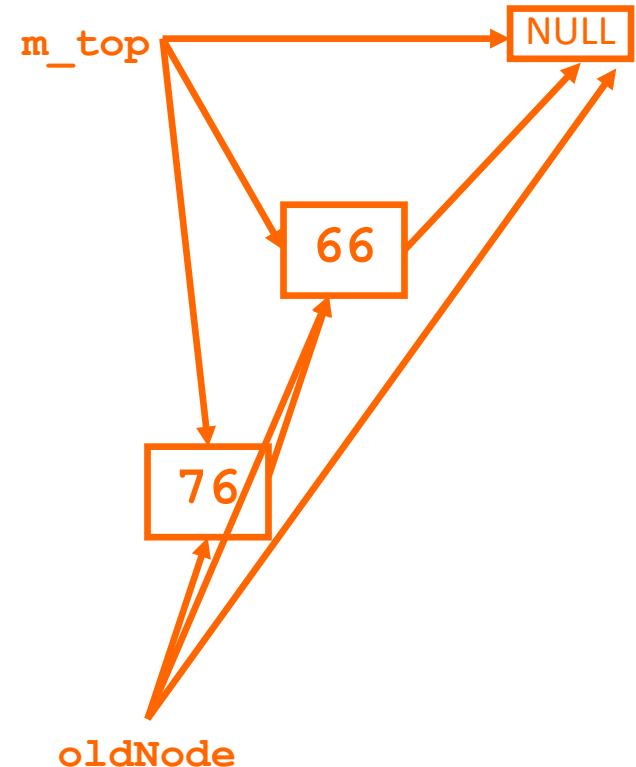
Linked List Push Algorithm

- PUSH (DataType data): boolean
- IF there is memory on the heap
- Get newNode from the heap
- Put data into the newNode
- IF m_top is NULL
- m_top = newNode
- ELSE
- newNode.next = m_top
- m_top = newNode
- ENDIF
- return TRUE
- ELSE
- return FALSE
- ENDIF
- END Push



Linked List Pop Algorithm

```
• POP (DataType data): boolean
  •   IF m_top == NULL
  •     return FALSE
  •   ELSE
  •     data = m_top.data
  •     oldNode = m_top
  •     m_top = oldNode.next
  •     release oldNode memory
  •     oldNode = NULL
  •     return TRUE
  •   ENDIF
  • END Pop
```



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 **Pop ()** and **Push ()** that it is supposed to have.
- Finally, there is the STL stack class, (requiring `<stack>`), which is obviously the **best** STL class to use your Stack class.
- STL stack is an adapted STL container (container adapter) for special use as a stack. No iterators are provided.
- However, even this must be encapsulated if it does not conform to our abstraction of what a stack should be (pointed out earlier and see slide notes from earlier). [1]

Features of the STL stack which don't fit in with our Abstraction

1. Its **pop ()** method, only removes the data, it does *not* pass it back to the calling method.
 2. In fact there is a **top ()** method which returns the data (by reference) at the top of the stack.
 3. Neither **pop ()** , **top ()** nor **push ()** return a boolean: overflow and underflow must be checked separately.
- Given the abstraction we are after, even the STL stack must be encapsulated.

Stack Header File using STL stack

- // Stack.h
- //
- // Stack class
- // Version
- // Nicola Ritter
- // modified smr
- //-----
- // NO I/O HERE. LET THE CLIENT DEAL WITH I/O
- #ifndef MY_STACK
- #define MY_STACK
- //-----
- #include <stack>
- #include <iostream>
- using namespace std;

```
• template <class DataType>
• class Stack
• {
• public:
•     Stack () {};
•     ~Stack () {};
•     bool Push(const DataType &data);
•     bool Pop (DataType &data);
•     bool Empty () const {return m_stack.empty();}
• private:
•     stack<DataType> m_stack; // encapsulated STL stack
• };
```



```
•      //-----
•      // It is a template, so we have to put all the code
•      // in the header file
•      //-----
•
•      template<class DataType>
•      bool Stack<DataType>::Push(const DataType &data)
•      {
•          bool okay = true;
•          try
•          {
•              m_stack.push(data);
•          }
•          catch (...)
•          {
•              okay = false;
•          }
•
•          return okay;
•      }
```

- //-----
- **template<class DataType>**
- **bool Stack<DataType>::Pop(DataType &data)**
- {
- **if (m_stack.size() > 0)**
- {
- **data = m_stack.top();**
- **m_stack.pop();**
- **return true;**
- }
- **else**
- {
- **return false;**
- }
- }
- //-----
- **#endif**

Simple Example of Stack Use

- // StackTest.cpp
- //
- // Tests Stack classes
- //
- // Nicola Ritter
- // Version 01
- // modified smr
- // Reverse a string
- //
- //-----
- #include <iostream>
- #include <string>
- #include "Stack.h" ← //Our stack
- using namespace std;

- `//-----`
- `typedef Stack<char> CharStack;`
 
- `void Input (string &str);`
- `void Reverse (const string &str, CharStack &temp);`
- `void Output (CharStack &temp); // const – check what it`
 `//does first??`
- `//-----`
- `int main()`
- `{`
- `string str;`
- `CharStack temp;`
- `Input (str);`
- `Reverse (str, temp); [1]`
- `Output (temp);`
- `cout << endl;`
- `return 0;`
- `}`

- //-----
- **void** Input (**string** &str)
- {
- **cout** << "Enter a string, then press <Enter>: ";
- **getline**(**cin**,str);
- }
- //-----
- **void** Reverse (**const string** &str, **CharStack** &temp)
- {
- **bool** okay = **true**;
- **for** (**int** index = 0; index < str.length() && okay; index++)
- {
- okay = temp.Push(str[index]);
- }
- }

- `//-----`
- `void Output (CharStack &temp) // would const work?`
- `{`
- `bool okay;`
- `char ch;`
- `cout << "Your string reversed is: ";`
- `okay = temp.Pop(ch);`
- `while (okay)`
- `{`
- `cout << ch;`
- `okay = temp.Pop(ch);`
- `}`
- `cout << endl;`
- `}`
- `//-----`

Screen Output

- Enter a string, then press <Enter>: This is a string
- Your string reversed is: gnirts a si sihT
- Press any key to continue . . .

Advantages of Implementations

- It is assumed for each of the containers below, that **our Stack** encapsulates it.

Array	Linked List	list/vector/deque	STL stack
Easy to code	Full memory control	Easy to code	Easier to code compared to all the others.
	Memory ‘never’ runs out.	Memory ‘never’ runs out.	Memory ‘never’ runs out.

Disadvantages of Implementations

- It is assumed for each of the containers below, that **our Stack** encapsulates it.

Array	Linked List	list/vector/deque	STL stack
Can run out of space easily.	More difficult to code as it uses pointers.	Excess code sitting ‘behind’ the implementation.	Excess code sitting ‘behind’ the implementation.
		Only available in with some languages. e.g C++ has STL, Java has Java collections framework	Only available with some languages. e.g. C++ has STL, Java has Java collections framework

Readings

- Textbook: Stacks and Queues, entire section on Stacks.
- For amore details of Stacks 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 stack is from the abstract stack. We want the abstract level – see earlier lecture notes on level of abstractions.
- Textbook: Standard Template Library, section on Container Adapters
- Library Ereserve: Deitel & Deitel, [C++ how to program \[ECMS\]. Chapter 15](#) part A. [1]



Murdoch
UNIVERSITY

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

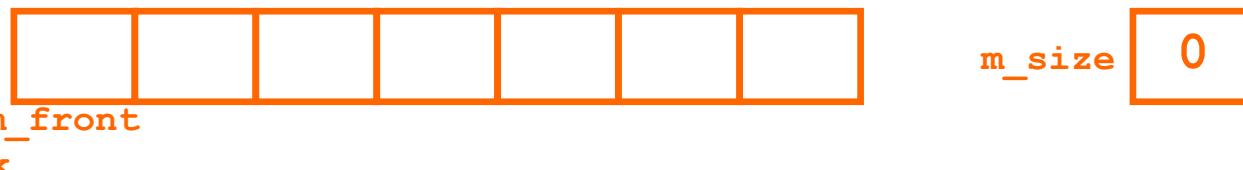
- Queues can be implemented any way you want, the encapsulation of the container used ensures that it does not matter.
- As long as it only has **Enqueue** , **Dequeue** , **Empty** and **Full** , then it is a minimal queue.
- Most commonly they are implemented using arrays, lists or an STL structure, 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)

Array Implementation



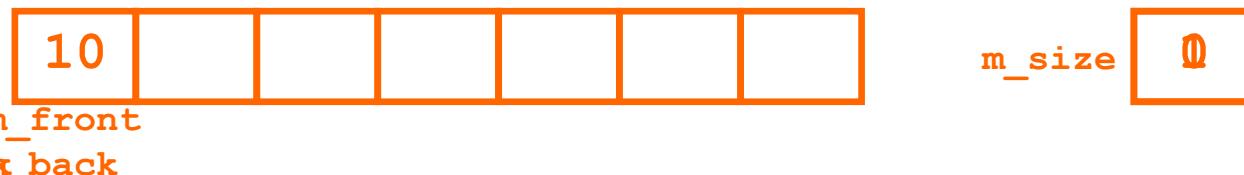
Linked List Implementation



Queue Example (Animation)

Enqueue (10)

Array Implementation



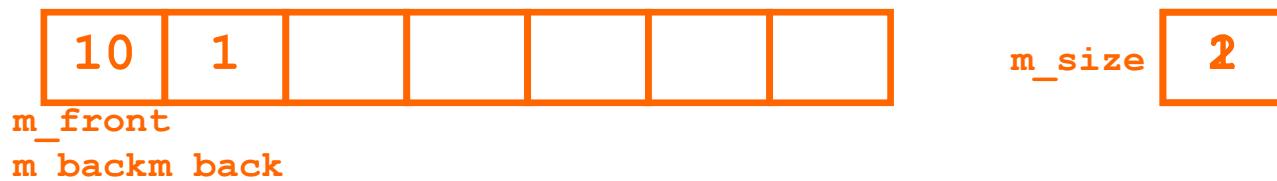
Linked List Implementation



Queue Example (Animation)

Enqueue (1)

Array Implementation



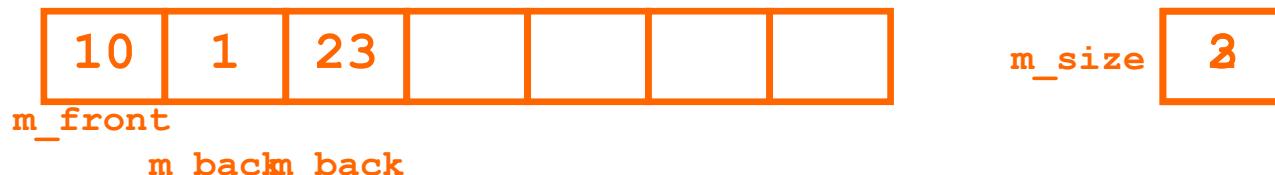
Linked List Implementation



Queue Example (Animation)

Enqueue (23)

Array Implementation



Linked List Implementation

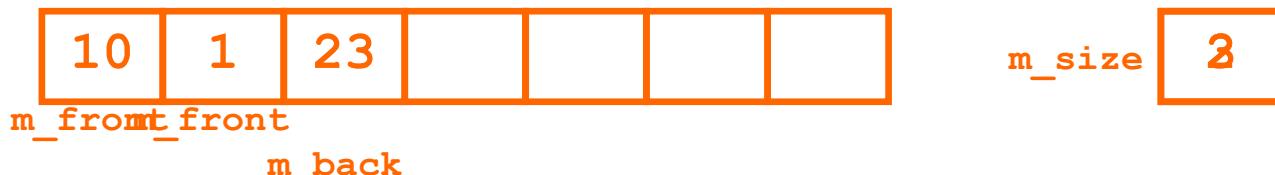


Queue Example (Animation)

Dequeue (num)

num 10

Array Implementation



Linked List Implementation

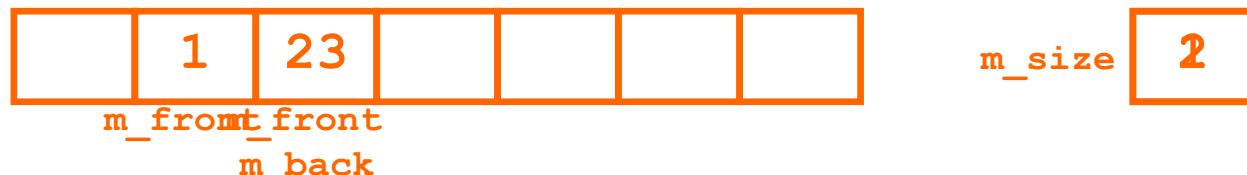


Queue Example (Animation)

Dequeue (num)

num 1

Array Implementation



Linked List Implementation

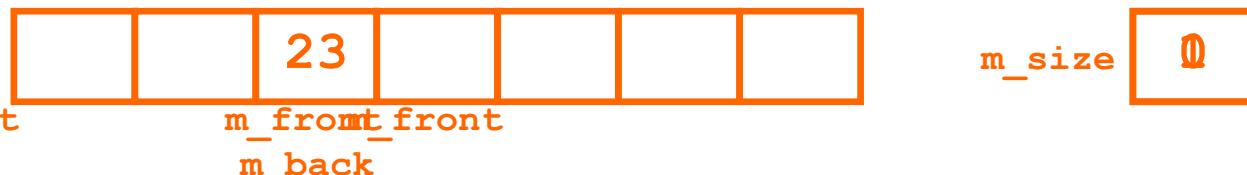


Queue Example (Animation)

Dequeue (num)

num 23

Array Implementation



Linked List Implementation

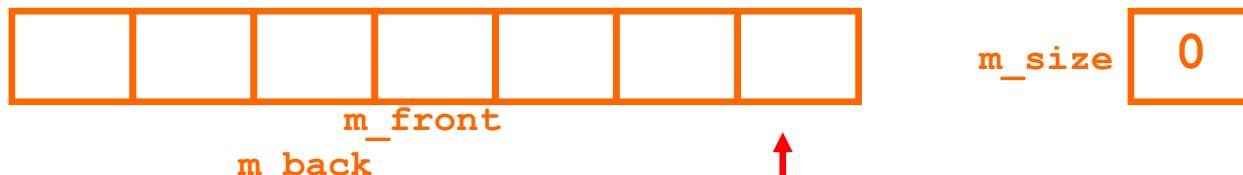


Queue Example (Animation)

Dequeue (num)

num

Array Implementation



Linked List Implementation

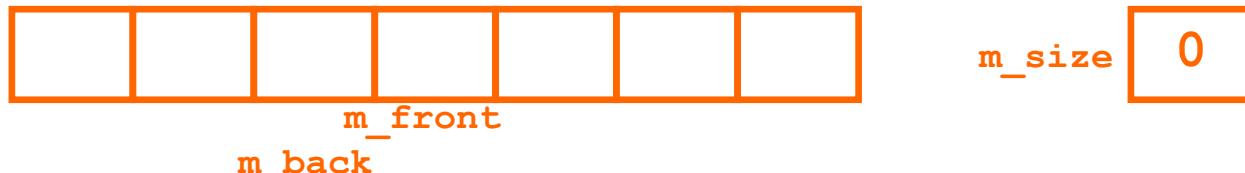


Queue Example (Animation)

Dequeue (num)

num

Array Implementation



Linked List Implementation



Queue Example (Animation)

Enqueue (12)

Array Implementation



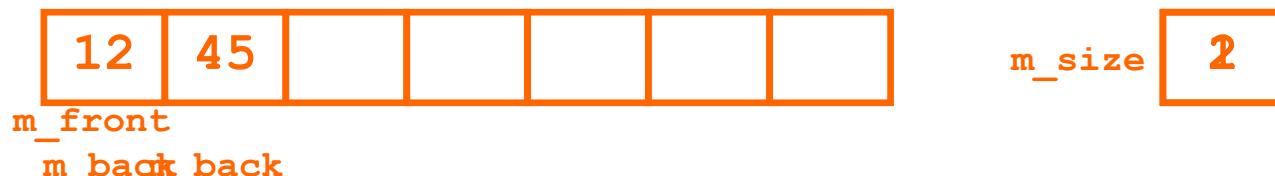
Linked List Implementation



Queue Example (Animation)

Enqueue (45)

Array Implementation



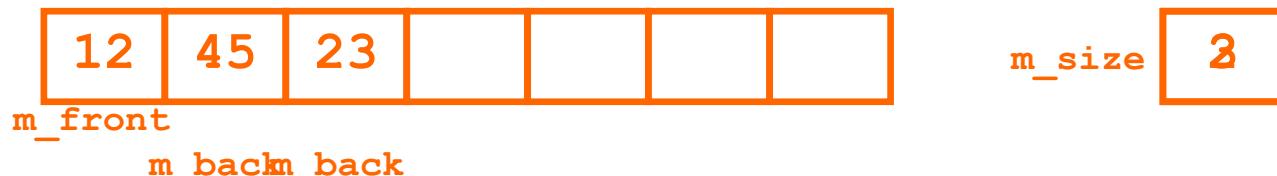
Linked List Implementation



Queue Example (Animation)

Enqueue (23)

Array Implementation



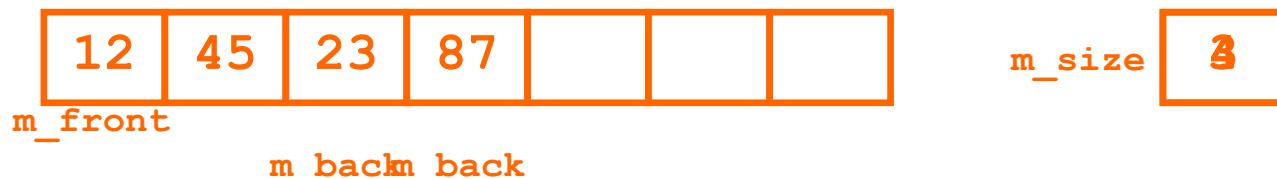
Linked List Implementation



Queue Example (Animation)

Enqueue (87)

Array Implementation



Linked List Implementation



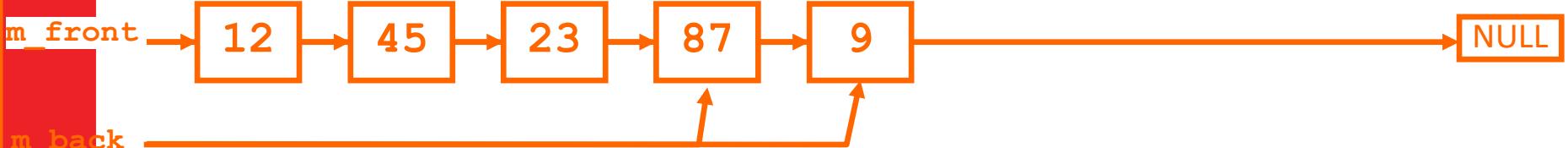
Queue Example (Animation)

Enqueue (9)

Array Implementation



Linked List Implementation



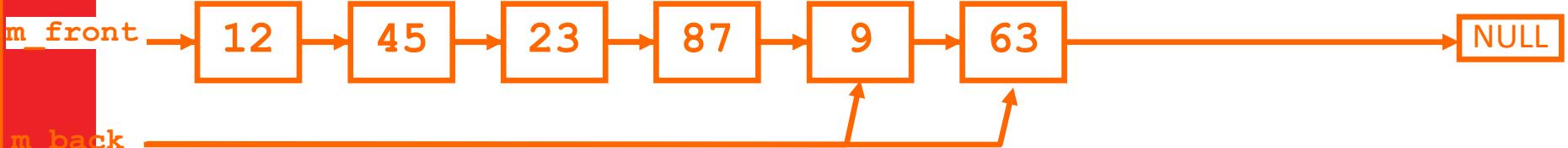
Queue Example (Animation)

Enqueue (63)

Array Implementation



Linked List Implementation



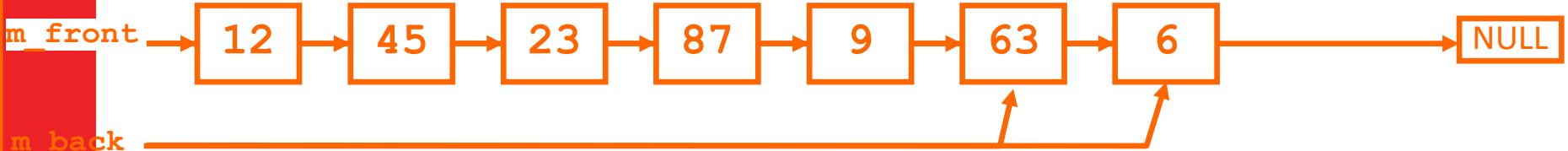
Queue Example (Animation)

Enqueue (6)

Array Implementation



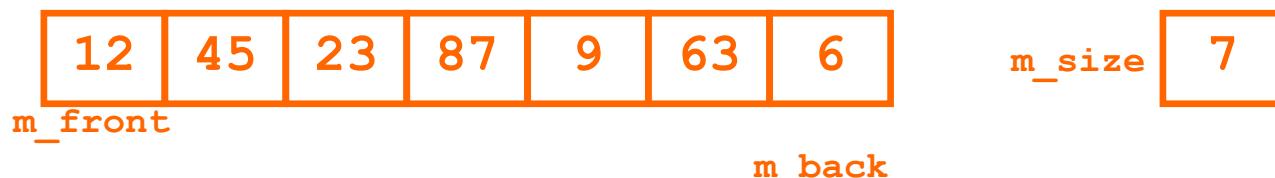
Linked List Implementation



Queue Example (Animation)

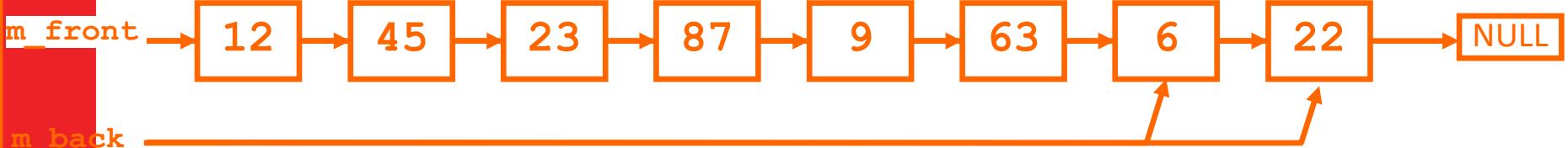
Enqueue (22)

Array Implementation



OVERFLOW

Linked List Implementation



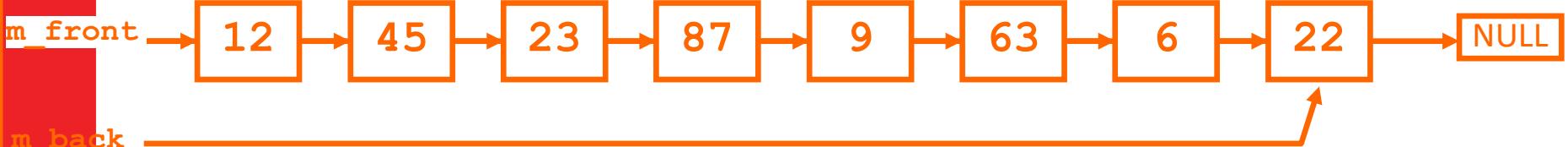
Queue Example (Animation)

Dequeue (num)

Array Implementation



Linked List Implementation



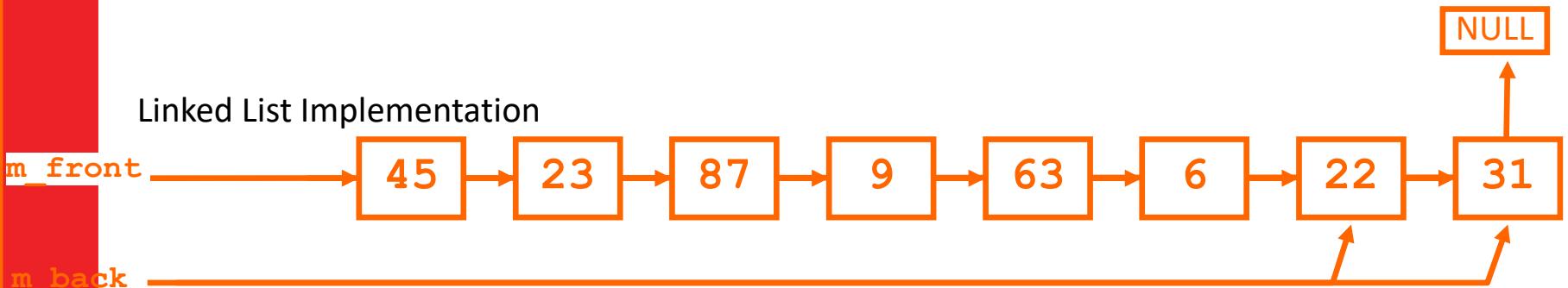
Queue Example (Animation)

Enqueue (31)

Array Implementation



Linked List Implementation



Array Enqueue Algorithm

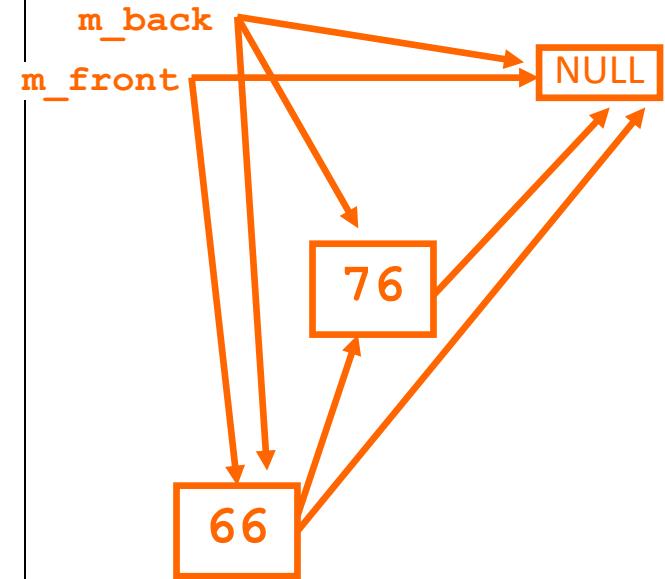
- ENQUEUE (DataType data): boolean
- IF $m_size \geq \text{ARRAY_SIZE}-1$
- return FALSE
- ELSE
- Increment m_size
- Increment $m_back \bmod \text{ARRAY_SIZE}$ [1]
- Place data at position m_back
- return TRUE
- ENDIF
- END Enqueue

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

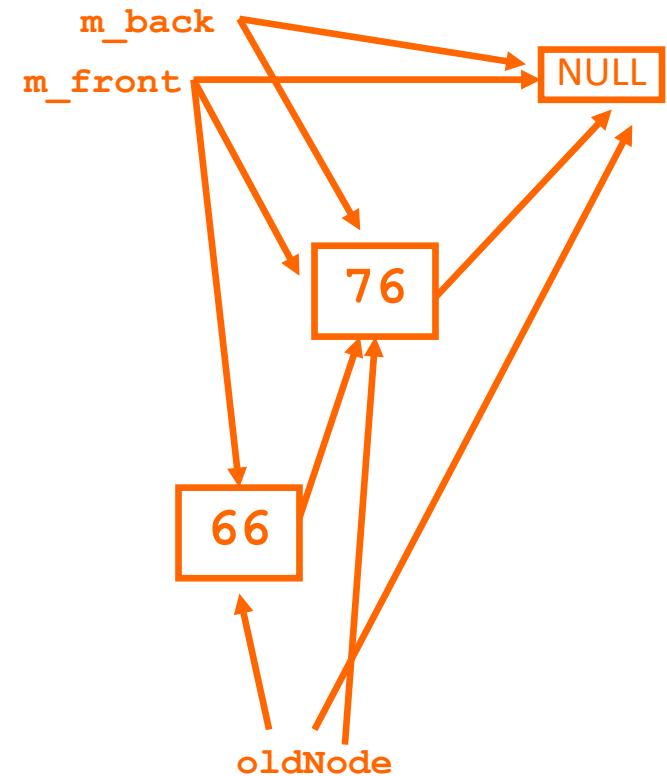
Linked List Enqueue Algorithm

- 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]

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]

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;
•  }
```

- //-----
- **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 &queue);`
- `void DeleteNumber (IntQueue &queue);`
- `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.  
• }  
  
• //-----
```

```
• 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 &queue)**
- {
- **int num;**
- **if (queue.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]

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 <code>m_back</code> ends up in front of <code>m_front</code>		Available in some languages like Java, C++. [1]	Available in some languages like Java, C++. [1]

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

Data Structures and Abstractions

Stack Example Animation

Lecture 23



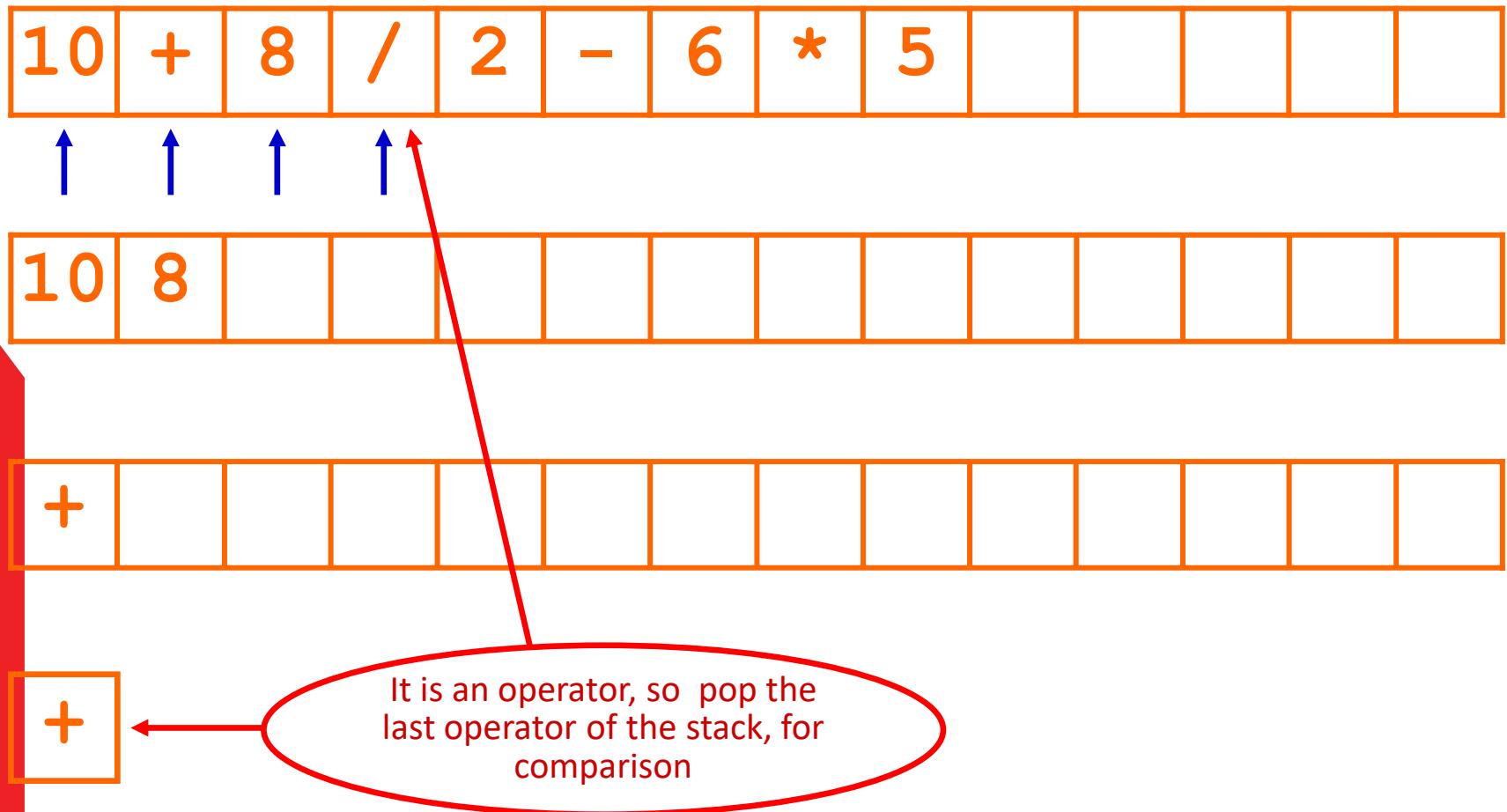
Stack Calculator Animation

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

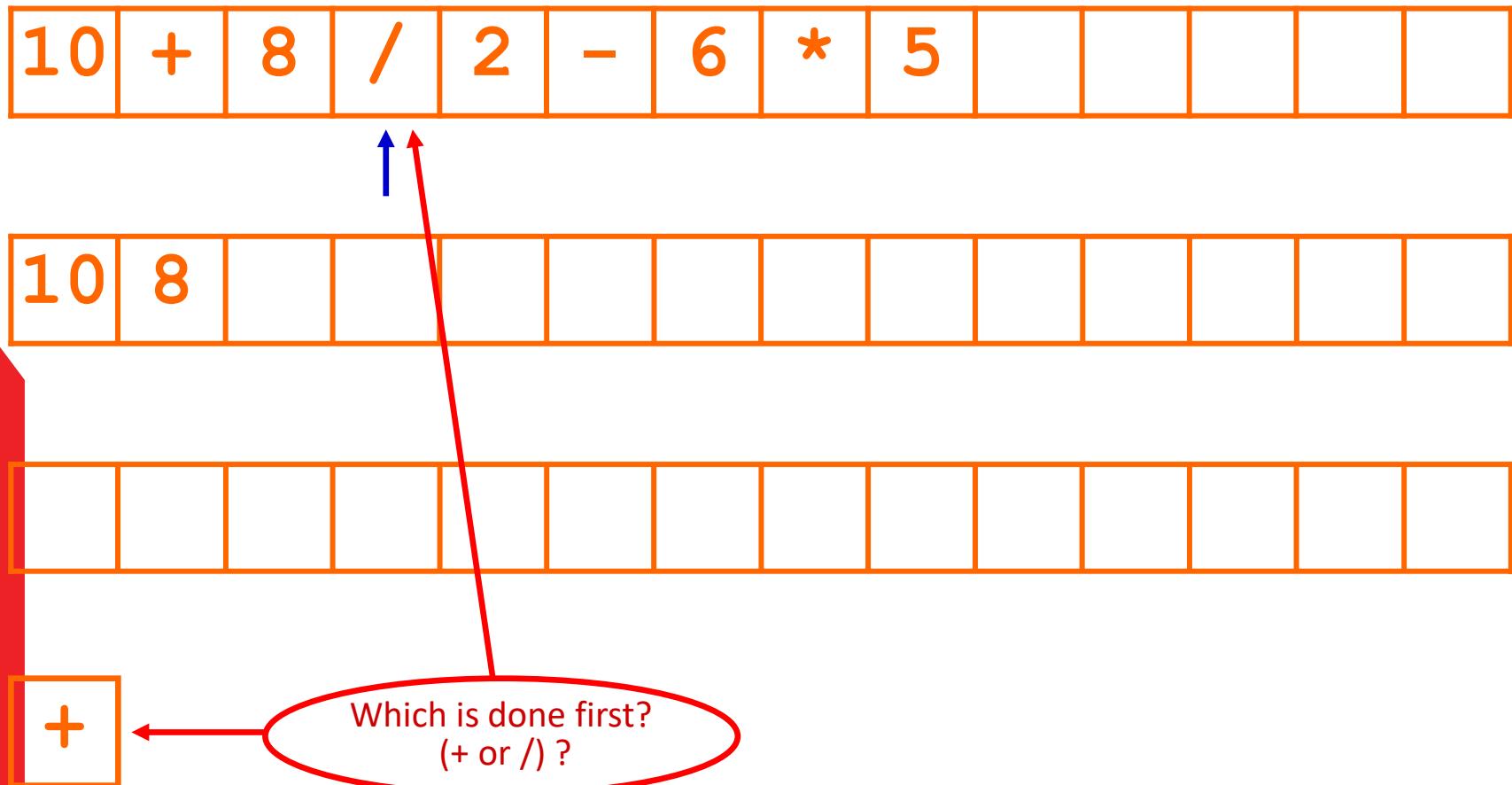
Stack Calculator Animation

10	+	8	/	2	-	6	*	5					
----	---	---	---	---	---	---	---	---	--	--	--	--	--

Stack Calculator Animation

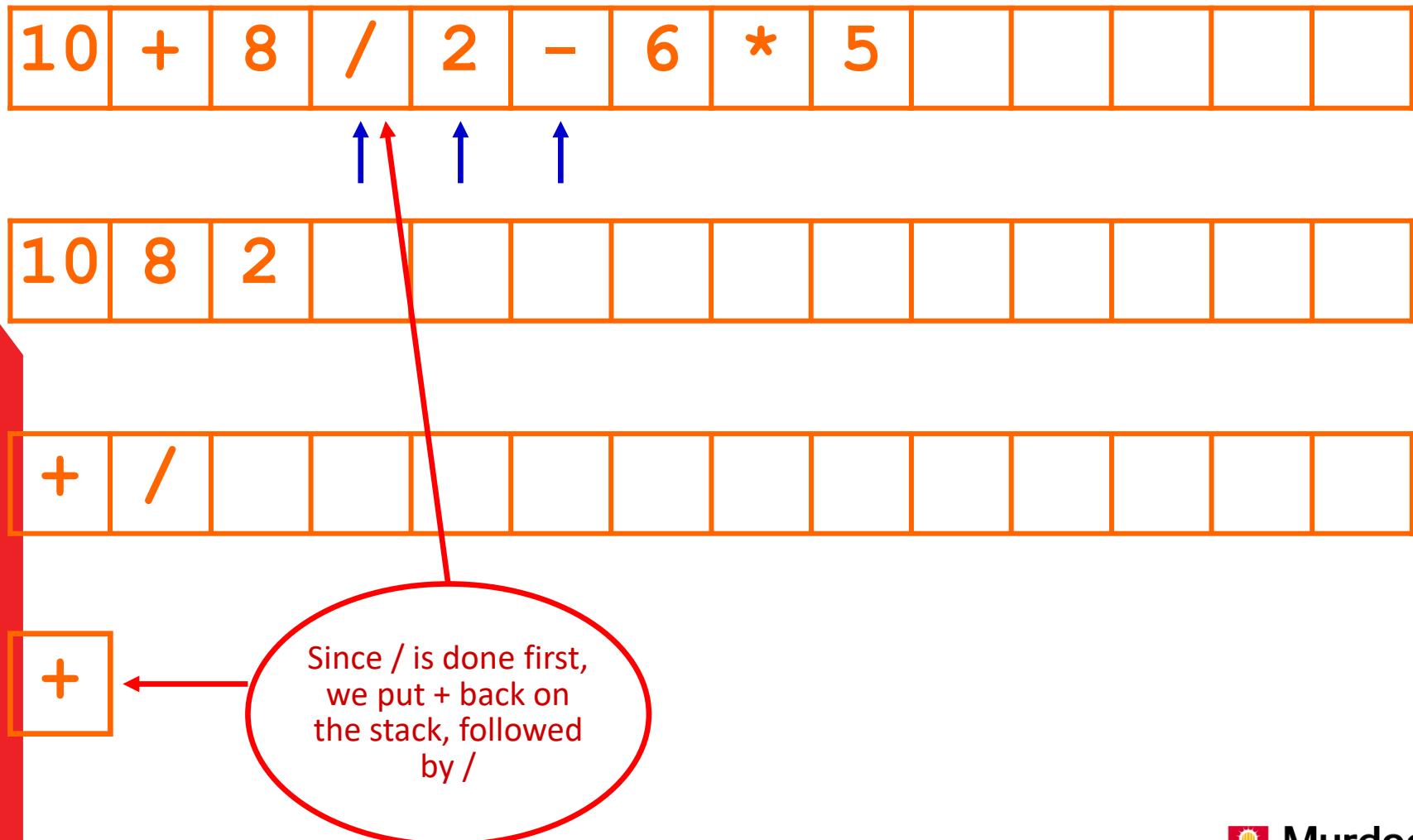


Stack Calculator Animation

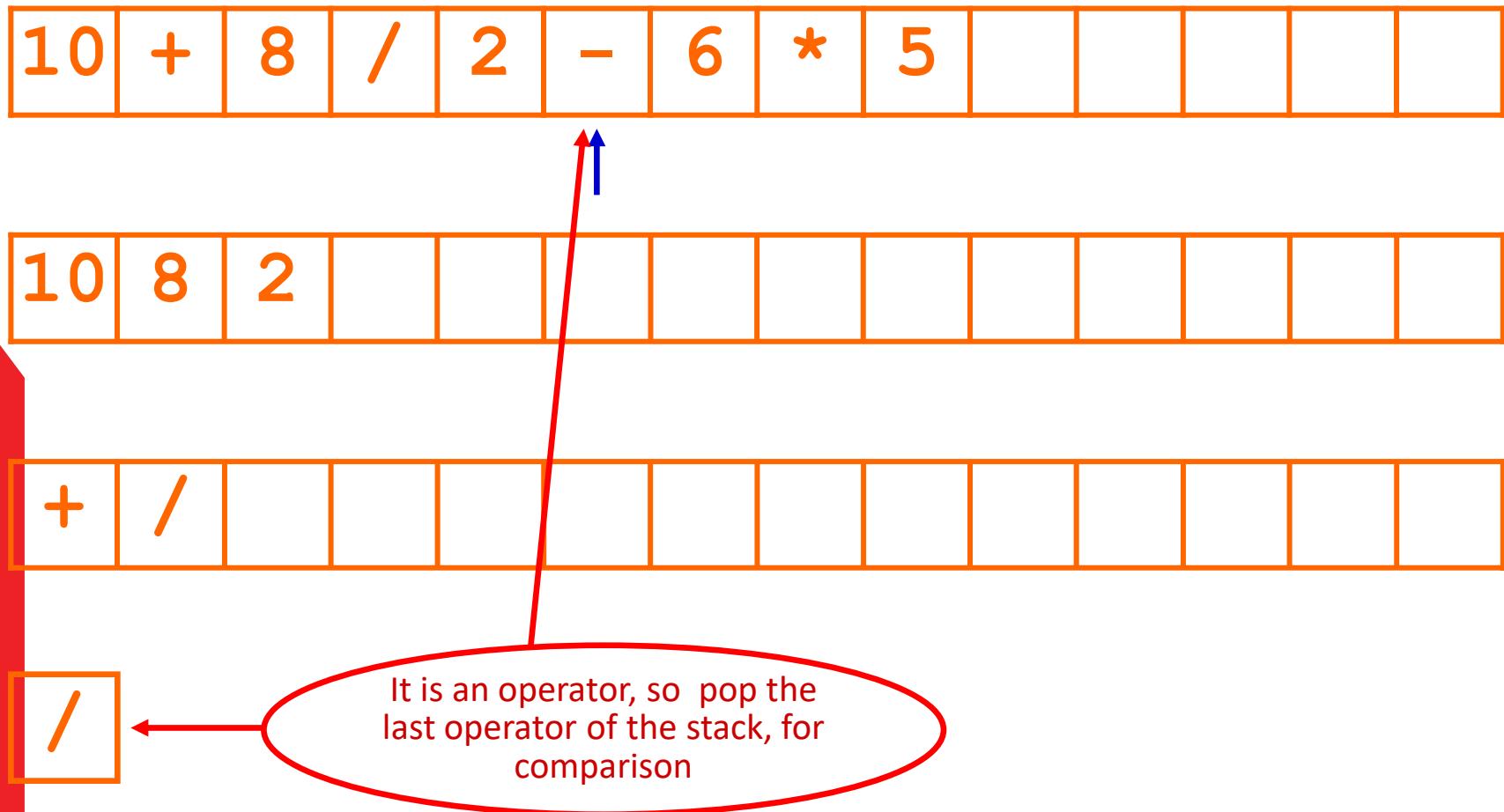


Which is done first?
(+ or /) ?

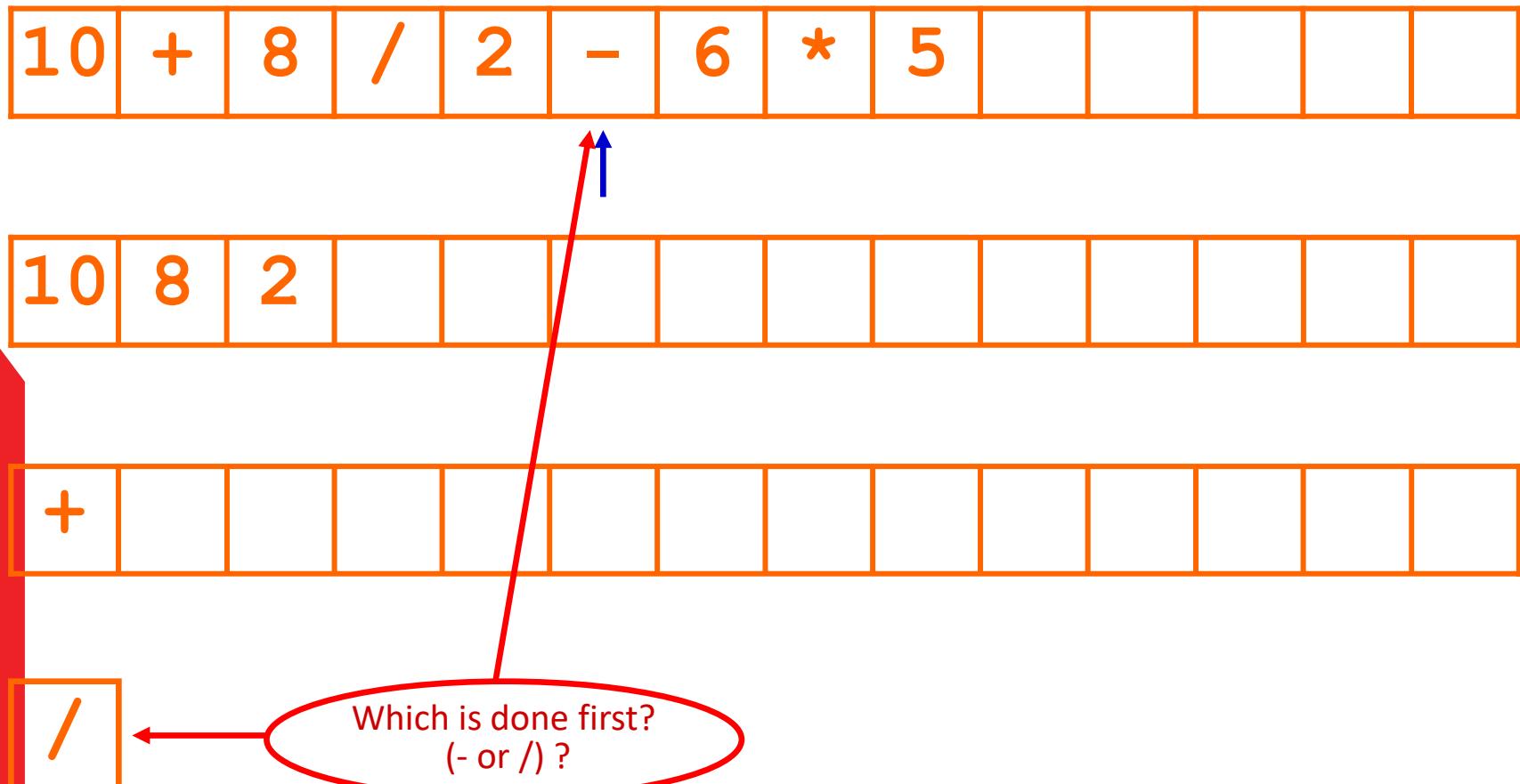
Stack Calculator Animation



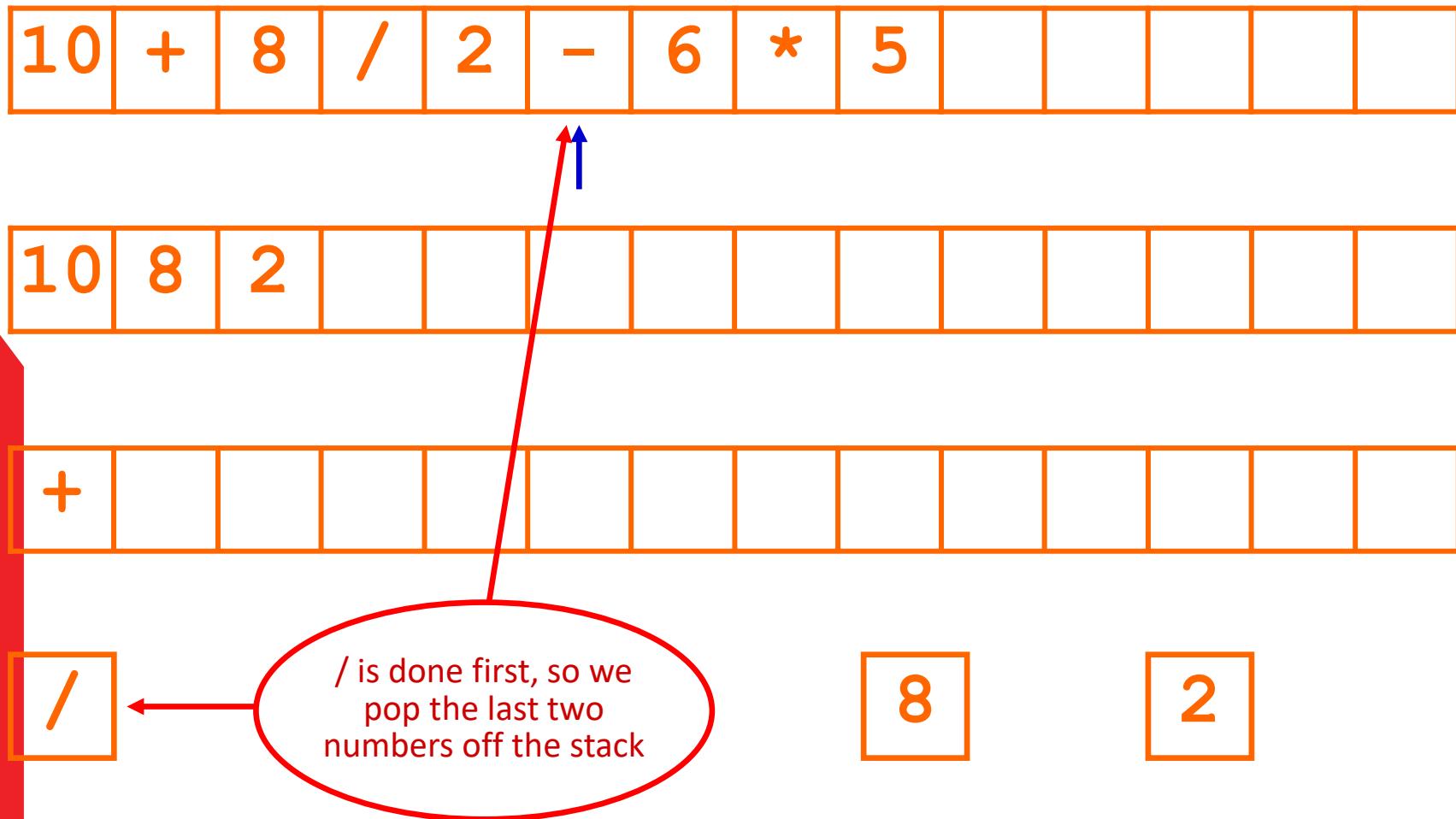
Stack Calculator Animation



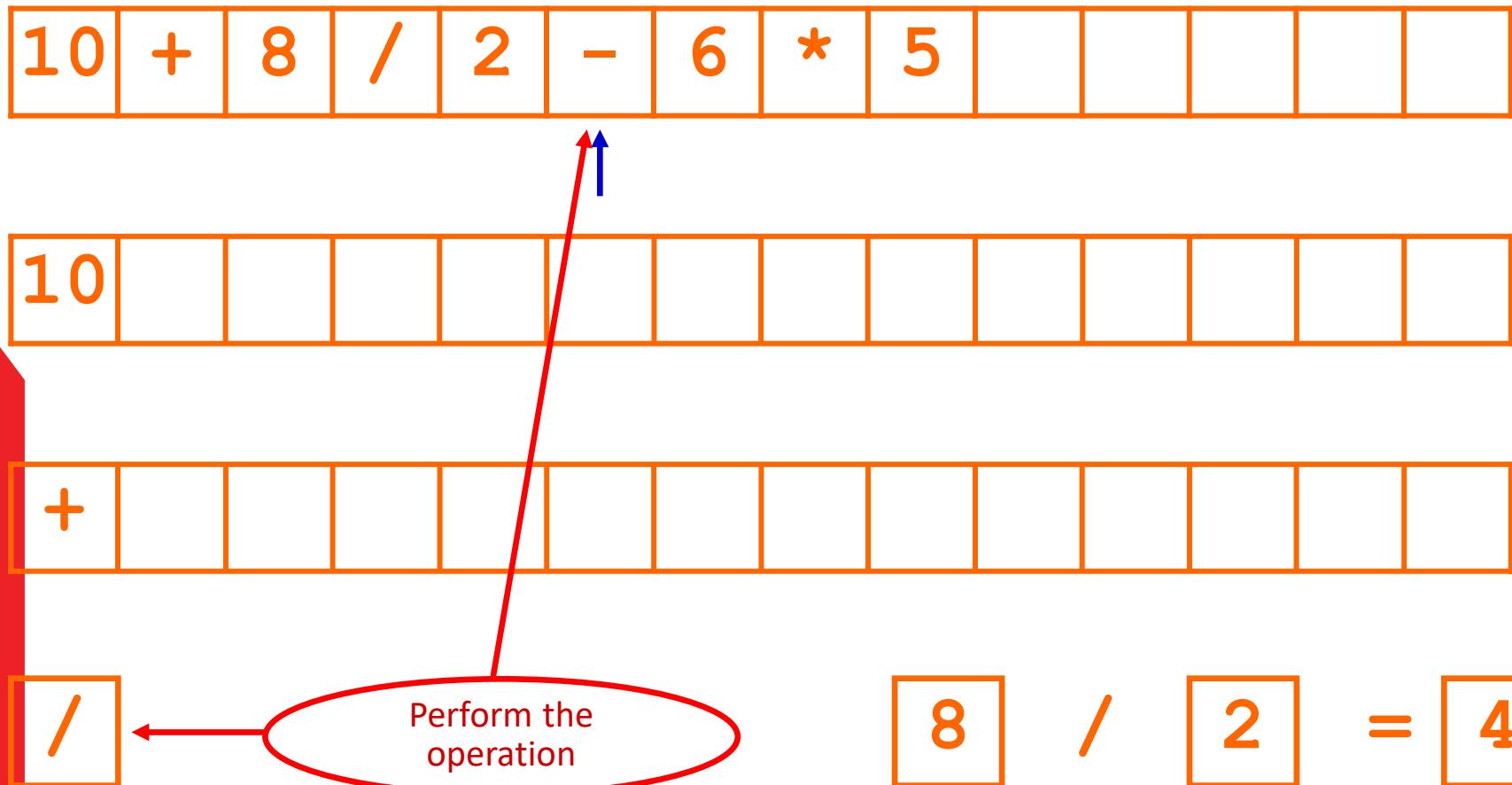
Stack Calculator Animation



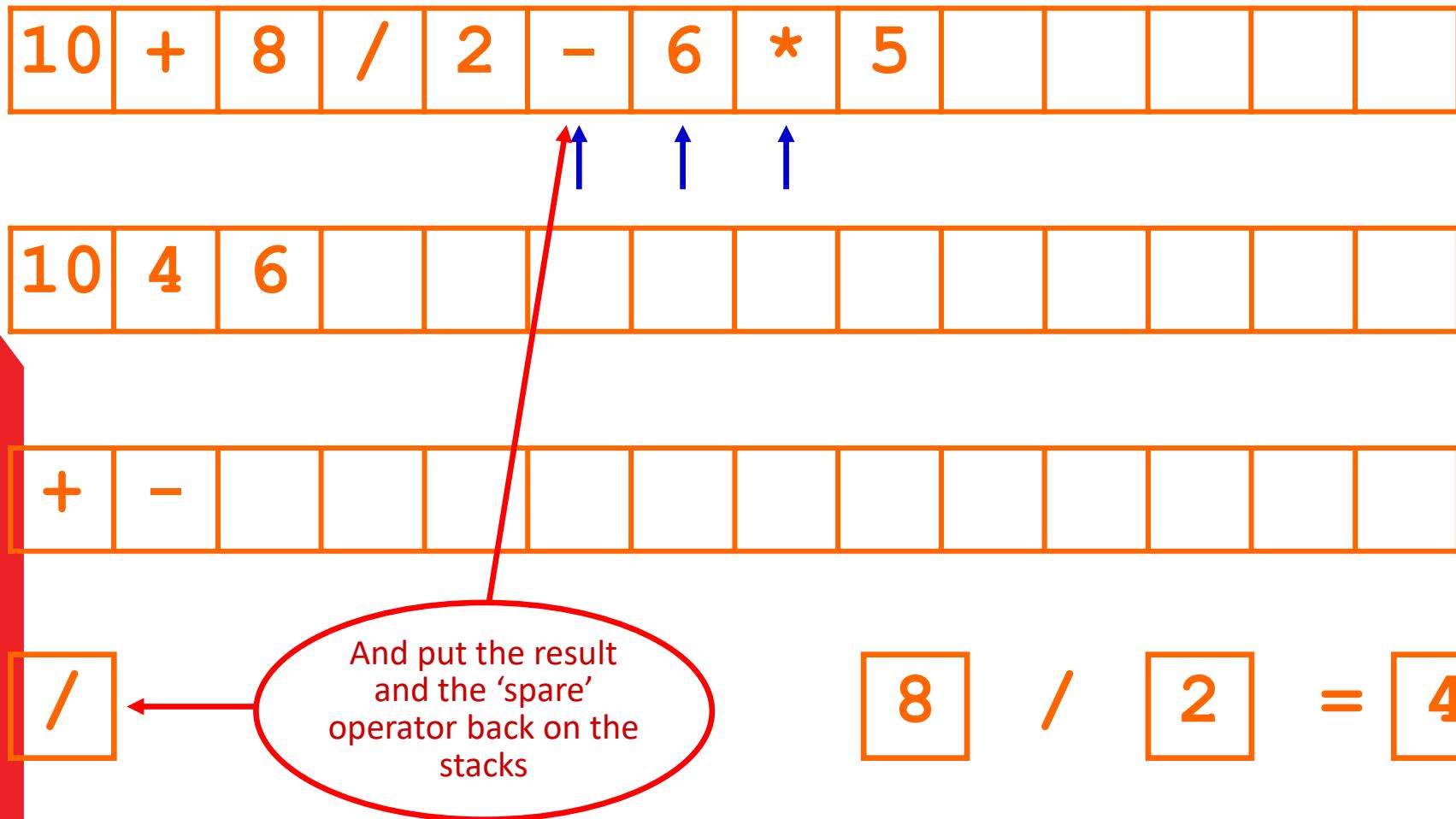
Stack Calculator Animation



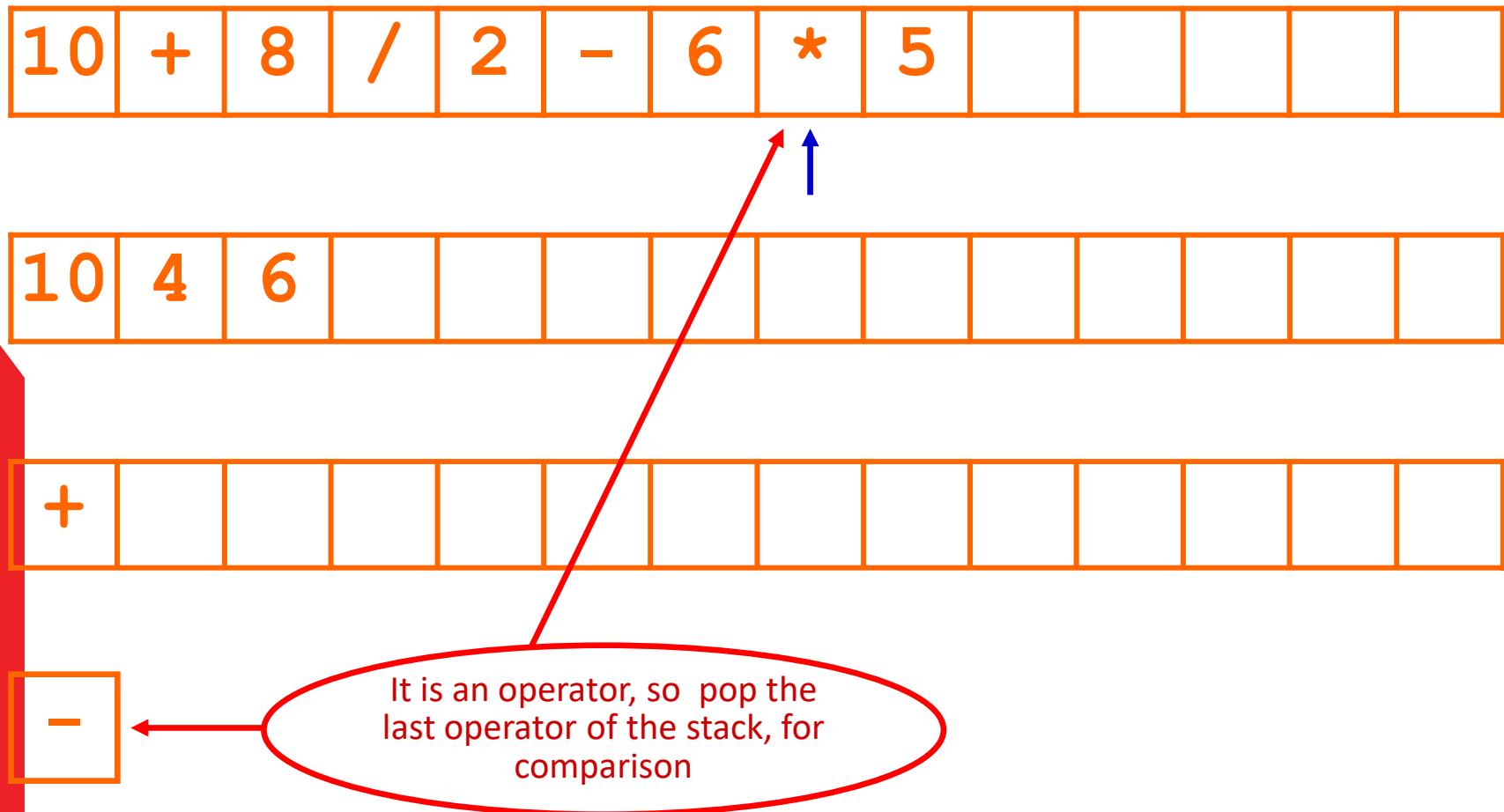
Stack Calculator Animation



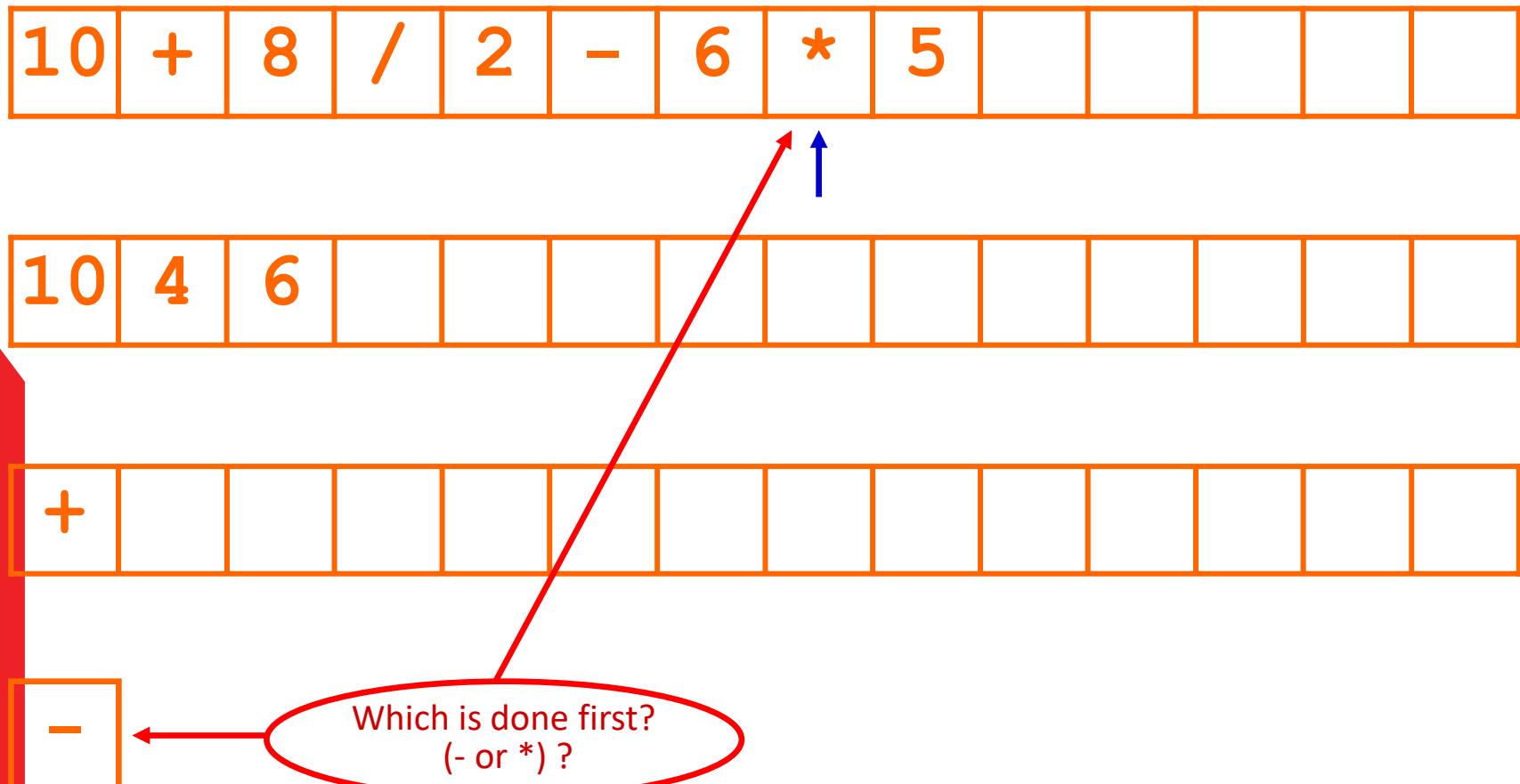
Stack Calculator Animation



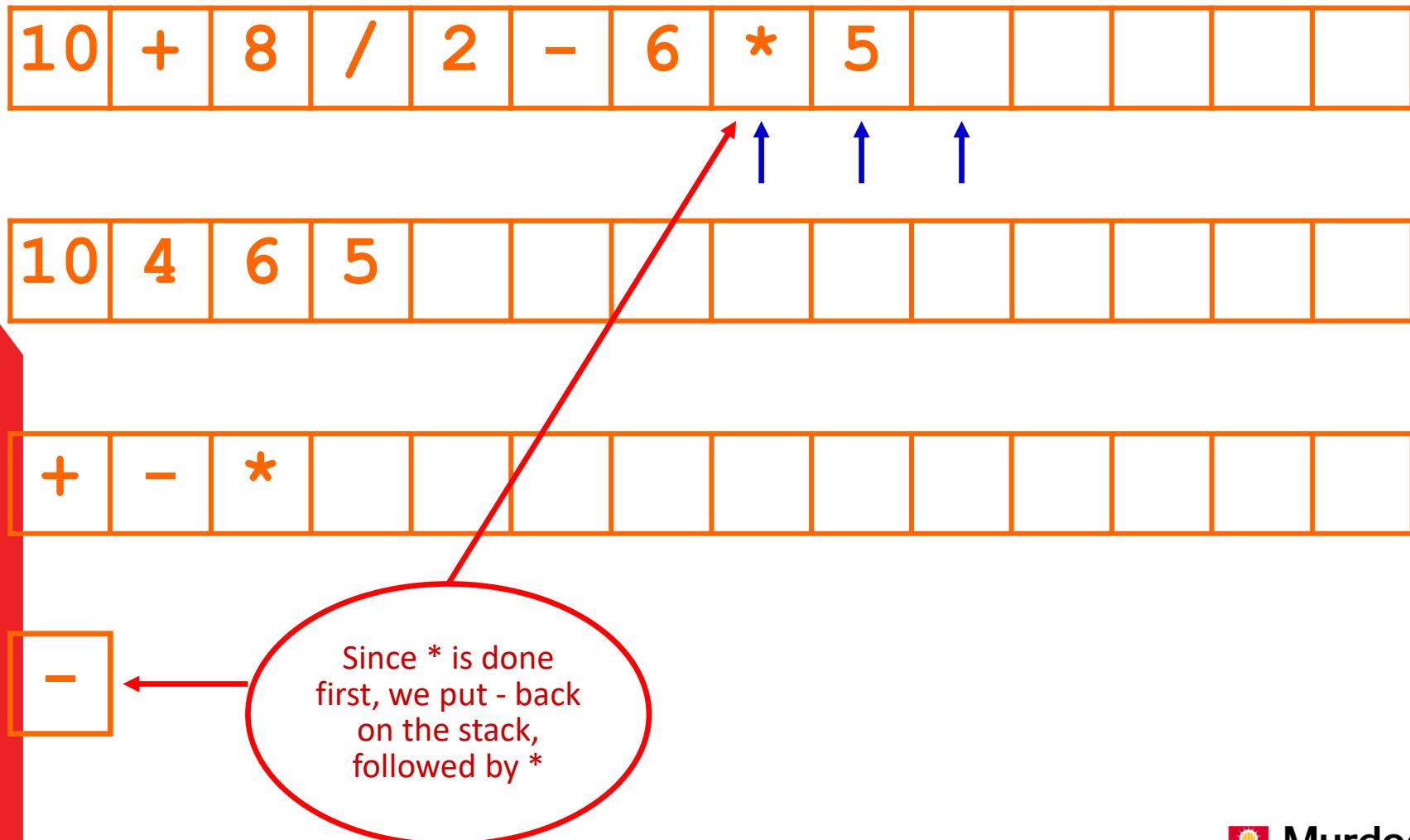
Stack Calculator Animation



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

Stack Calculator Animation



Stack Calculator Animation



A horizontal stack of 15 boxes representing a stack calculator. The first 9 boxes contain the expression: - (in a box), 4 (in a box), -, 30 (in a box), = (in a box), -26 (in a box). The 10th box is empty.

Stack Calculator Animation





A horizontal stack of 15 boxes. The first box contains the token +. The remaining 14 boxes are empty.

10 + -26 = -16

Stack Calculator Animation



+

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.

-26 = -16



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 logician Jan Lukasiewicz) notation (prefix notation), there is no need to use (). **Prefix** notation: $+ 2 3$
- Someone else came along later with something called **Reverse Polish Notation** (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

Data Structures and Abstractions

Searching, Merging and Sorting

Lecture 25



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]**

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.

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
```

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]

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

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.

- 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.

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
---	---	---	---	---

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.

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.

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
Selection	Merge	$n \log n$	$n \log n$
	Straight Selection	n^2	n^2
	Heap	$n \log n$	$n \log n$

* Based on empirical evidence only.

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.

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

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

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.

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



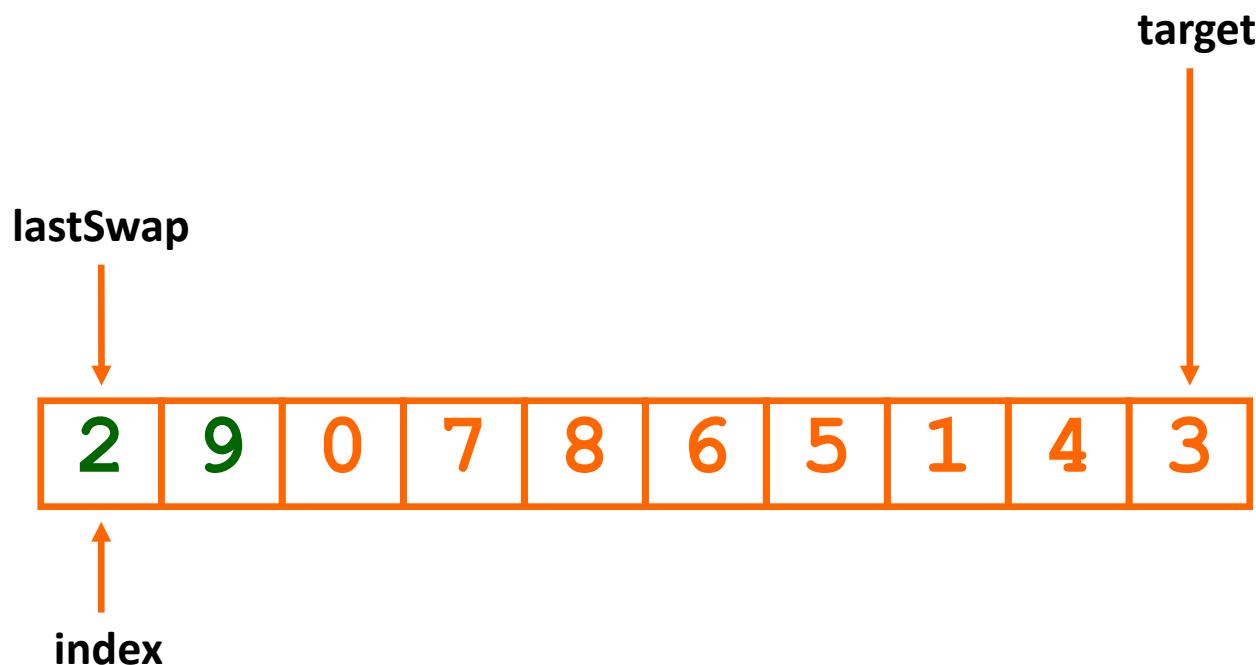
Bubble Sort

- 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.

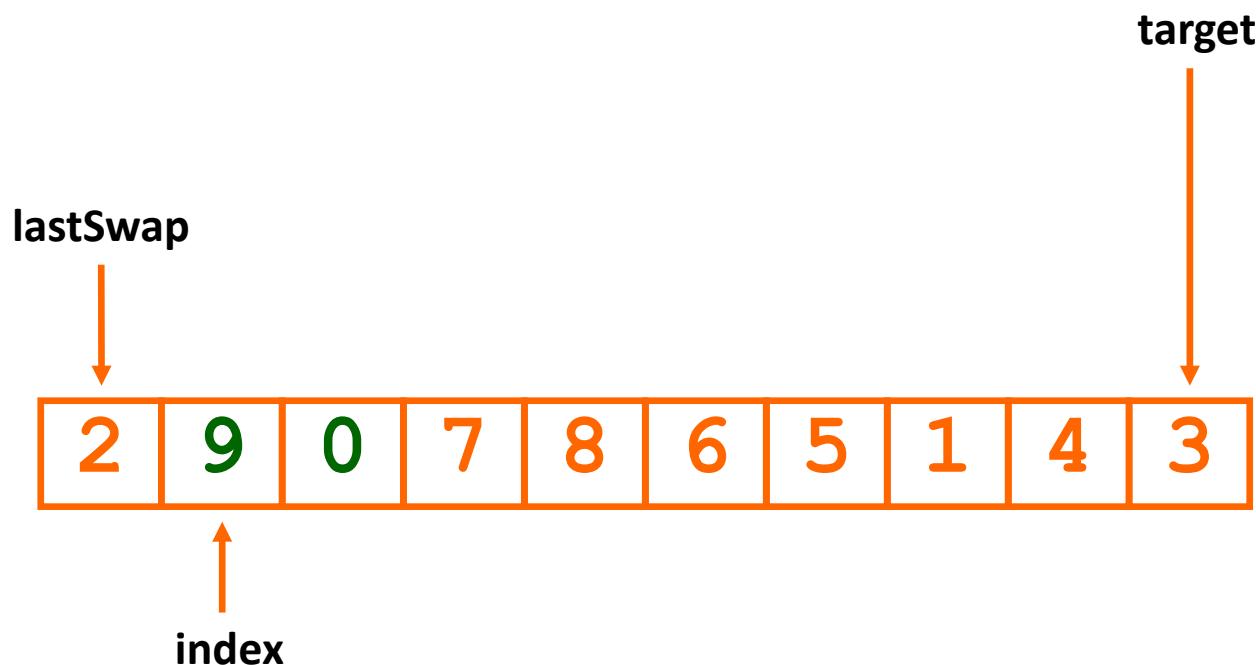
Bubble Sort Algorithm

- `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`

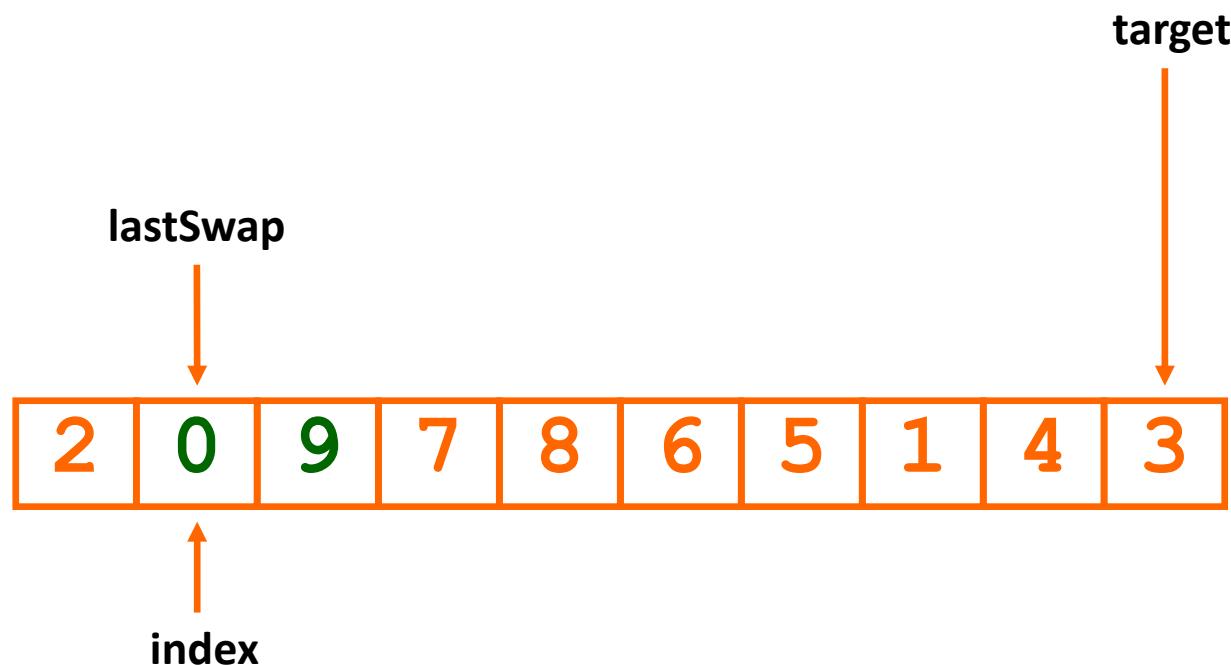
Bubble Sort Animation



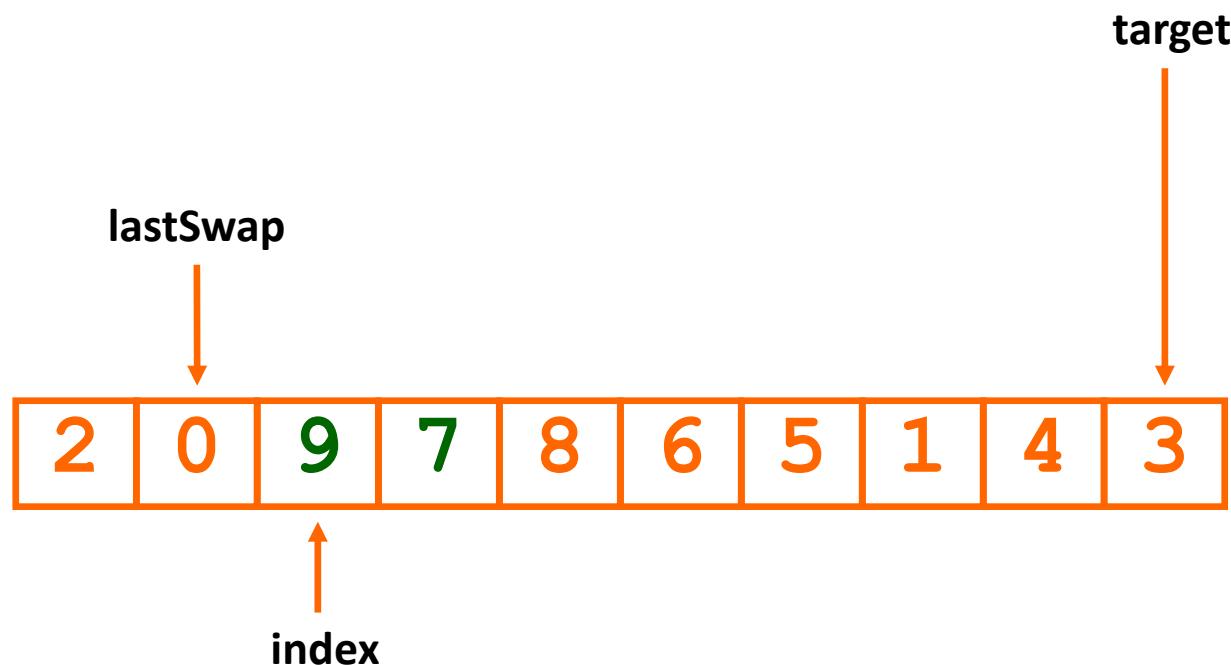
Bubble Sort Animation



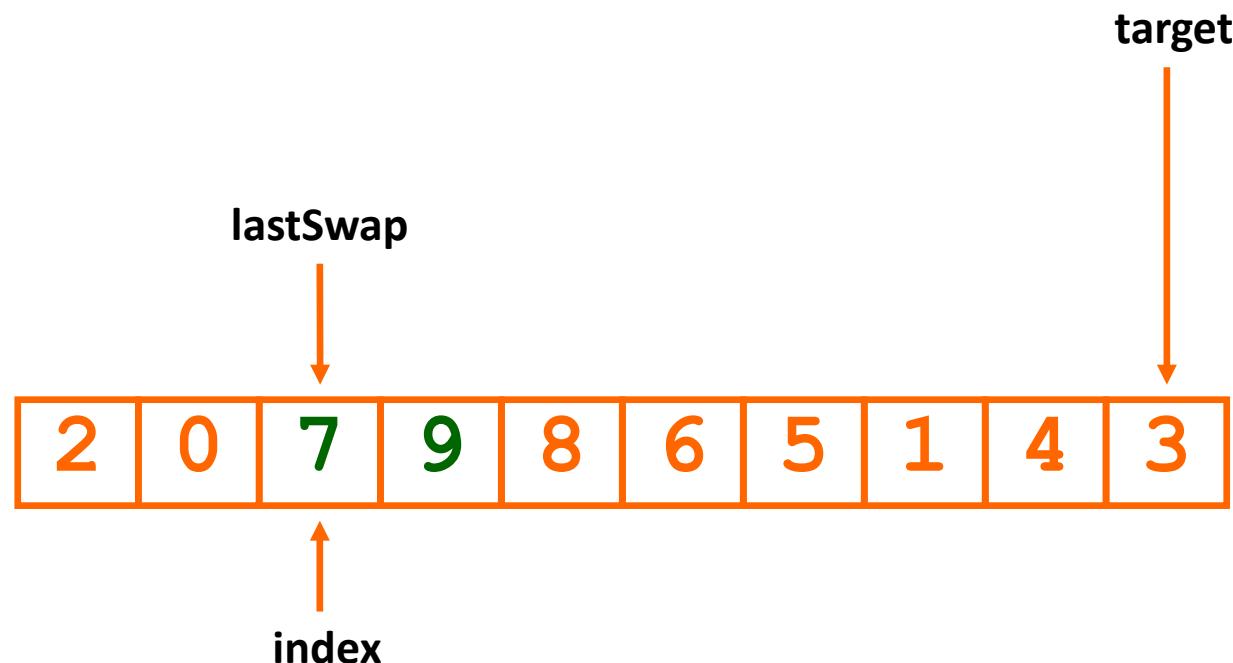
Bubble Sort Animation



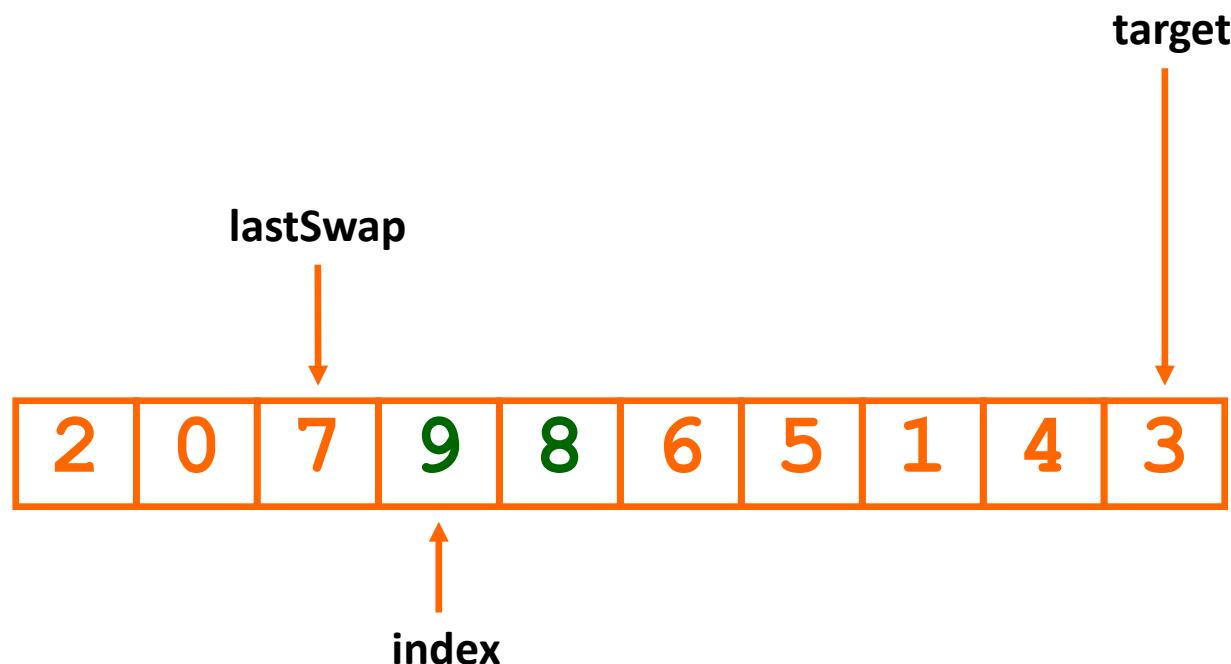
Bubble Sort Animation



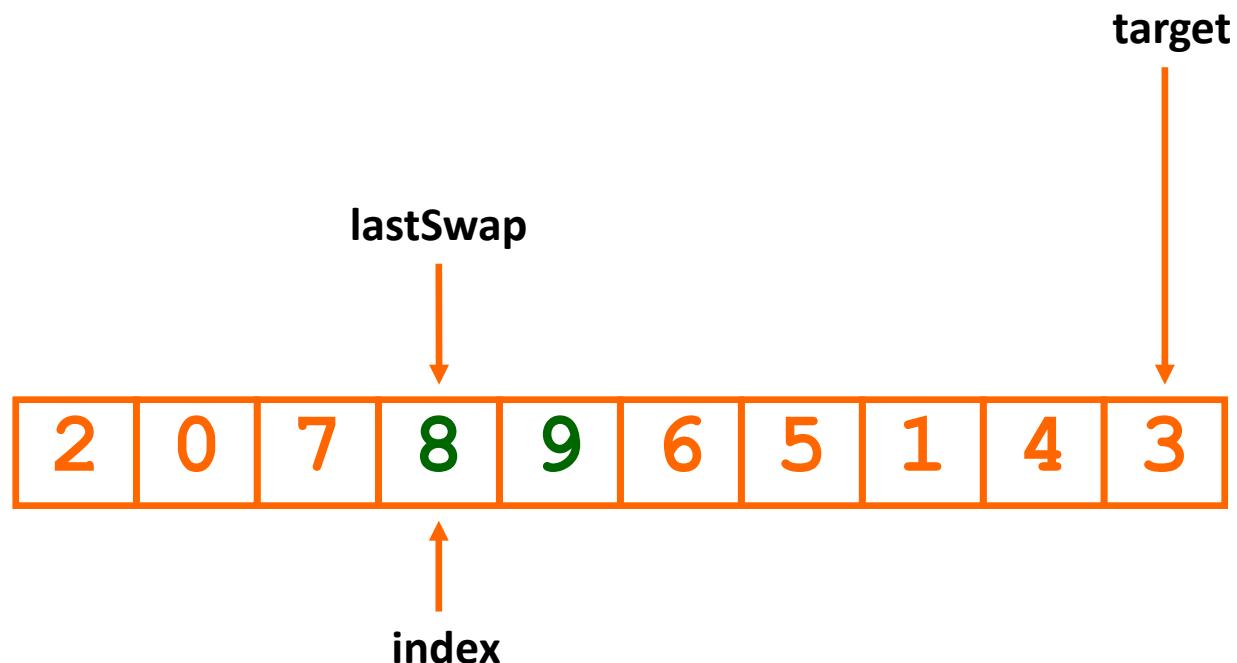
Bubble Sort Animation



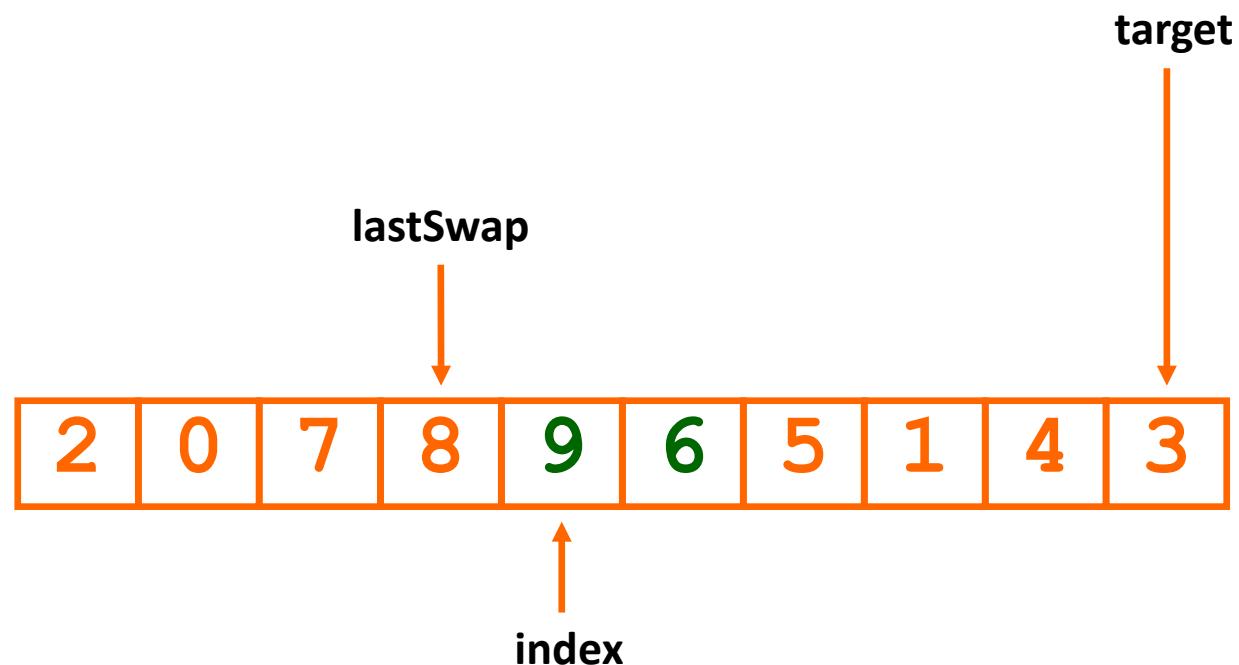
Bubble Sort Animation



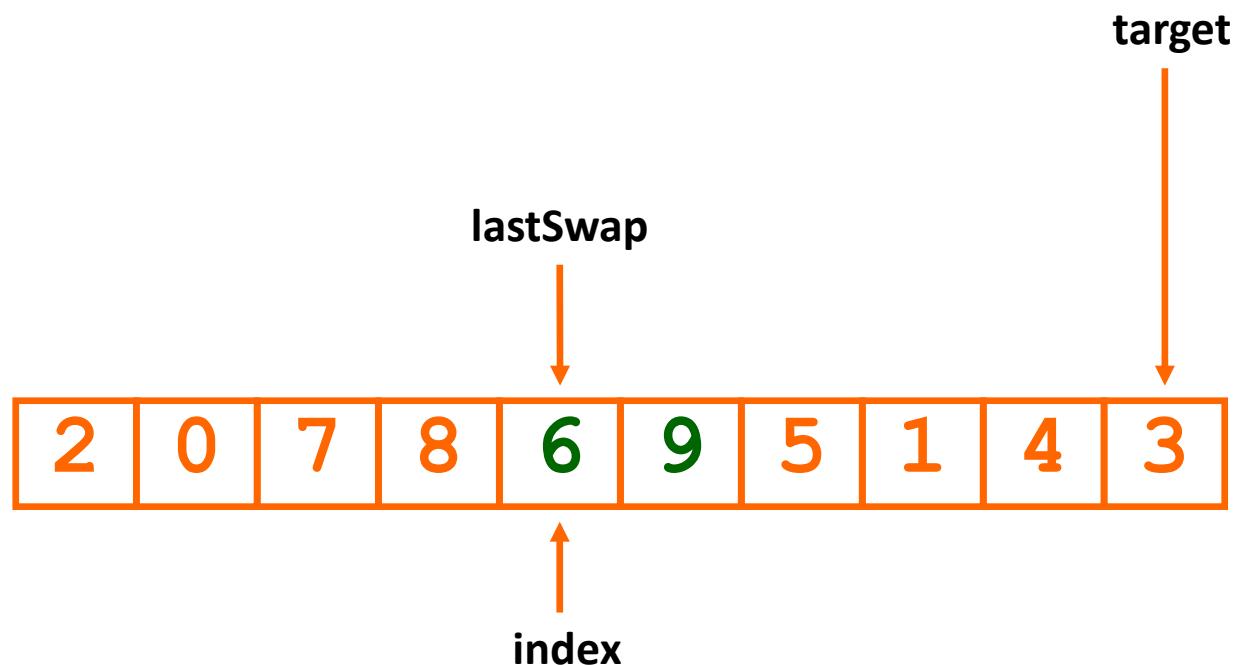
Bubble Sort Animation



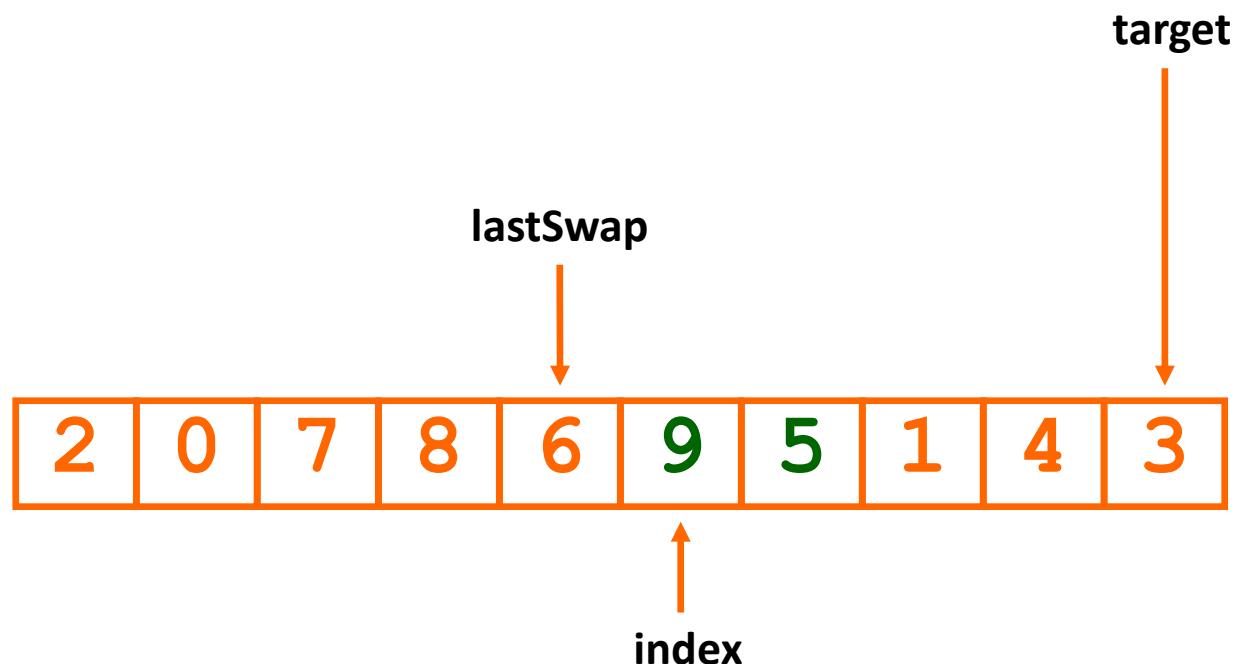
Bubble Sort Animation



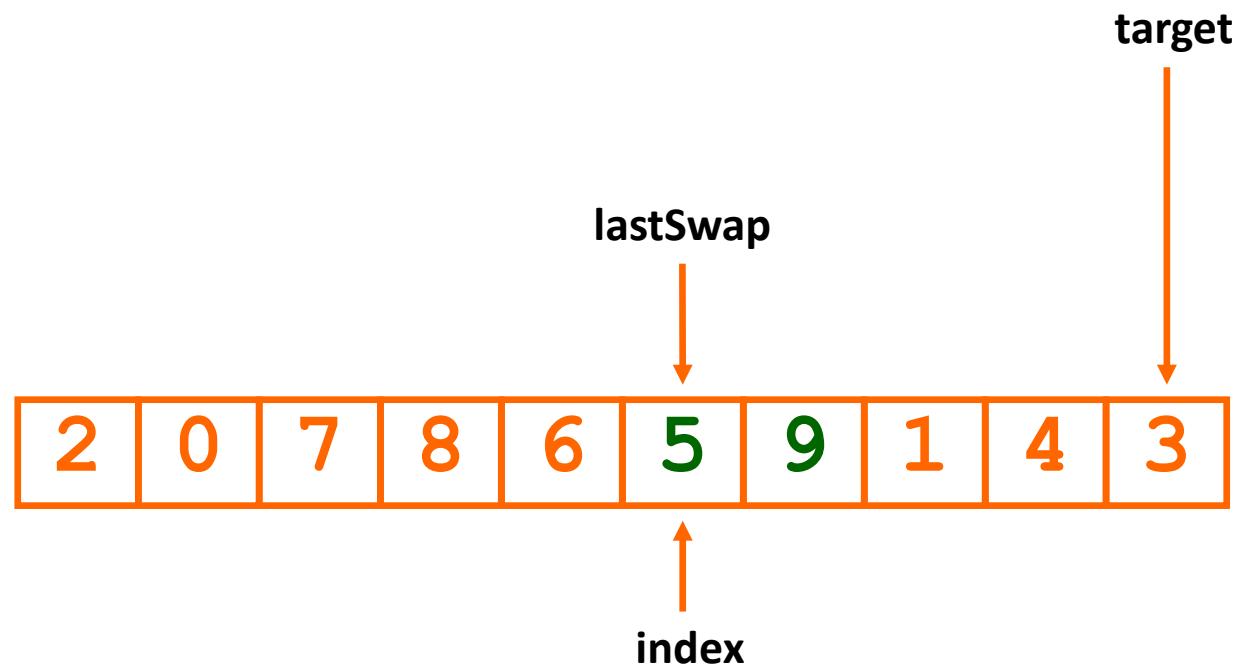
Bubble Sort Animation



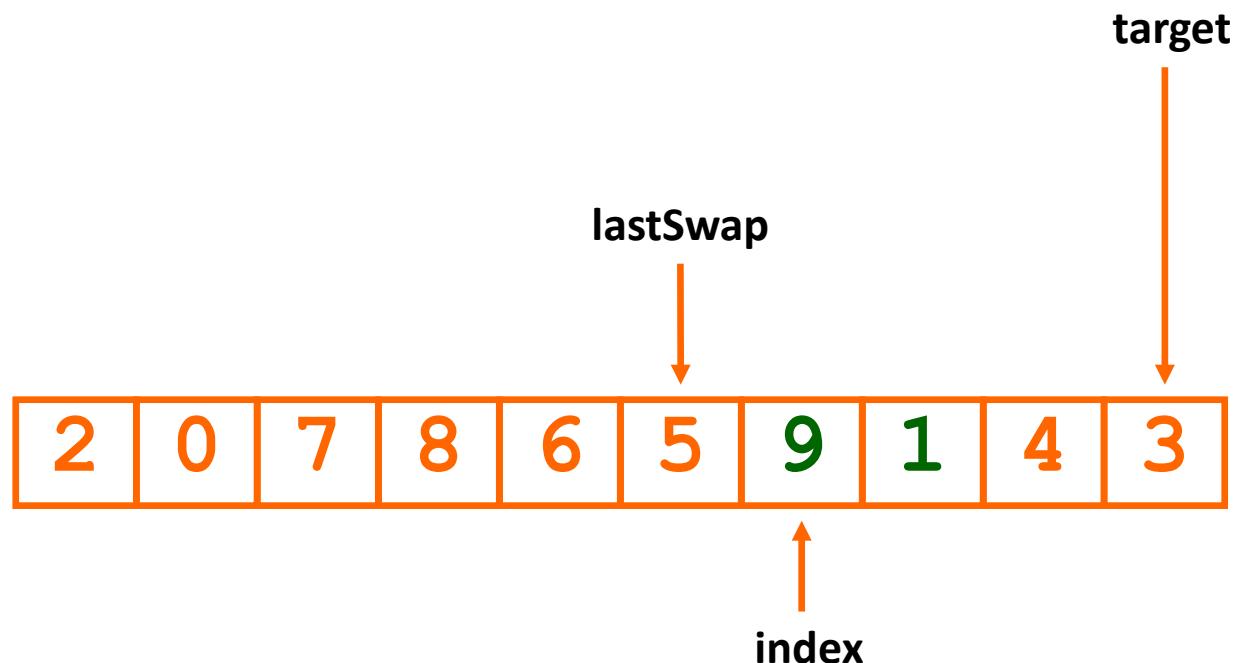
Bubble Sort Animation



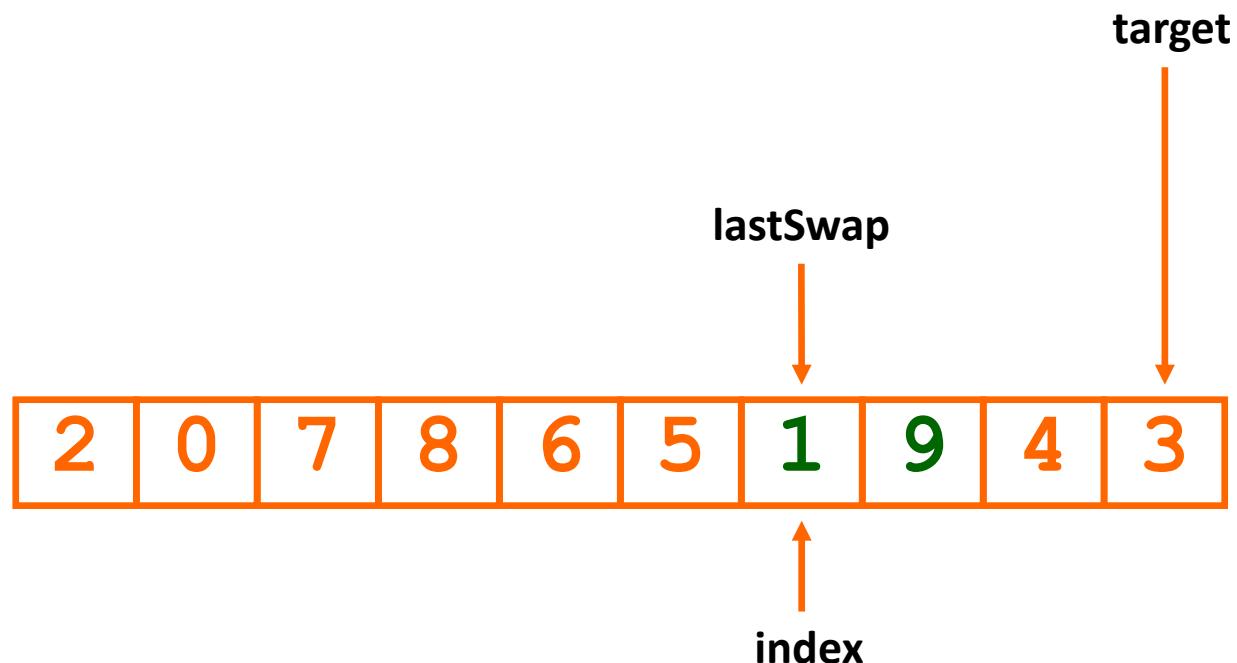
Bubble Sort Animation



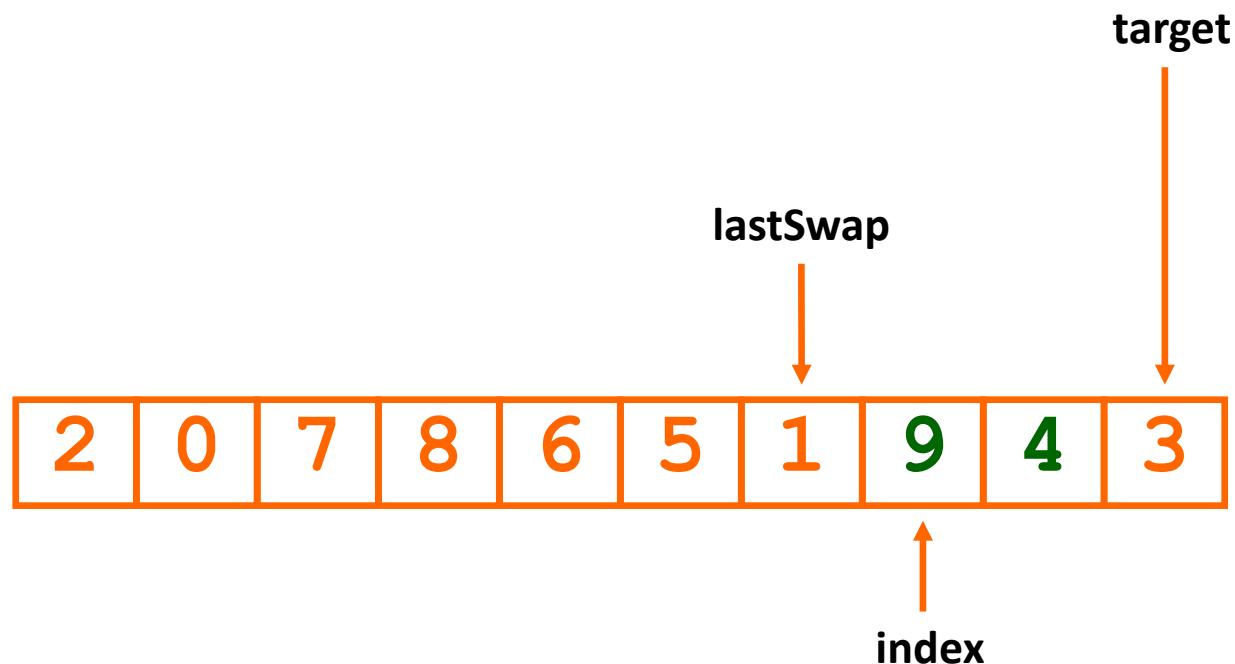
Bubble Sort Animation



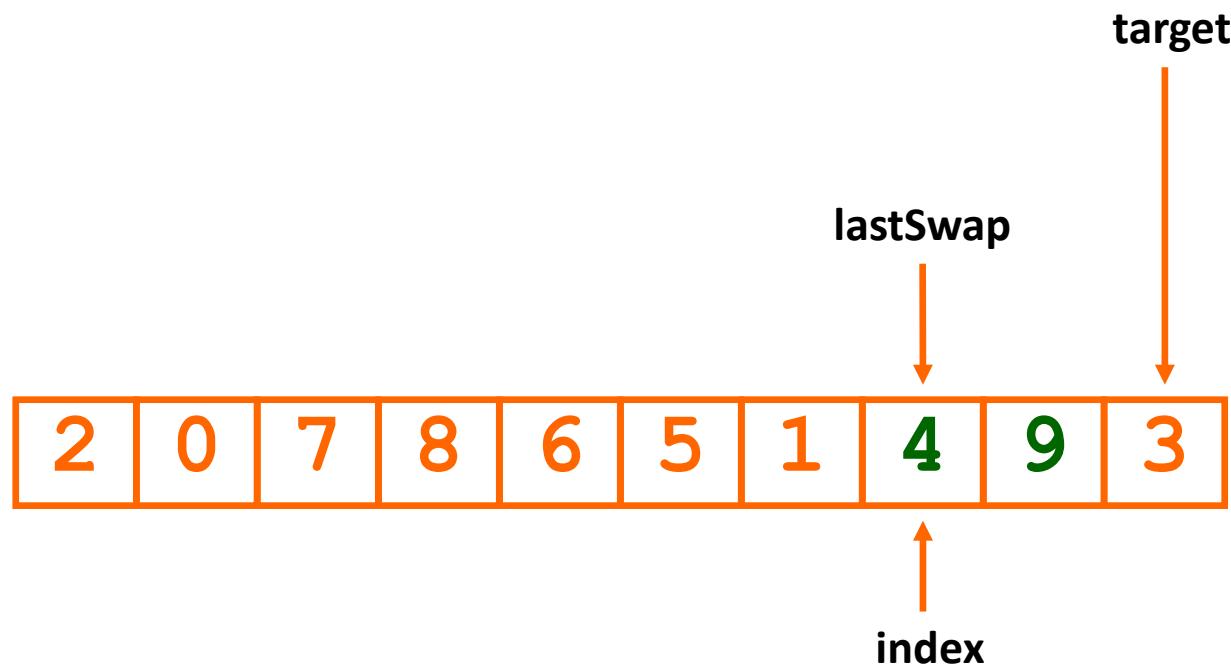
Bubble Sort Animation



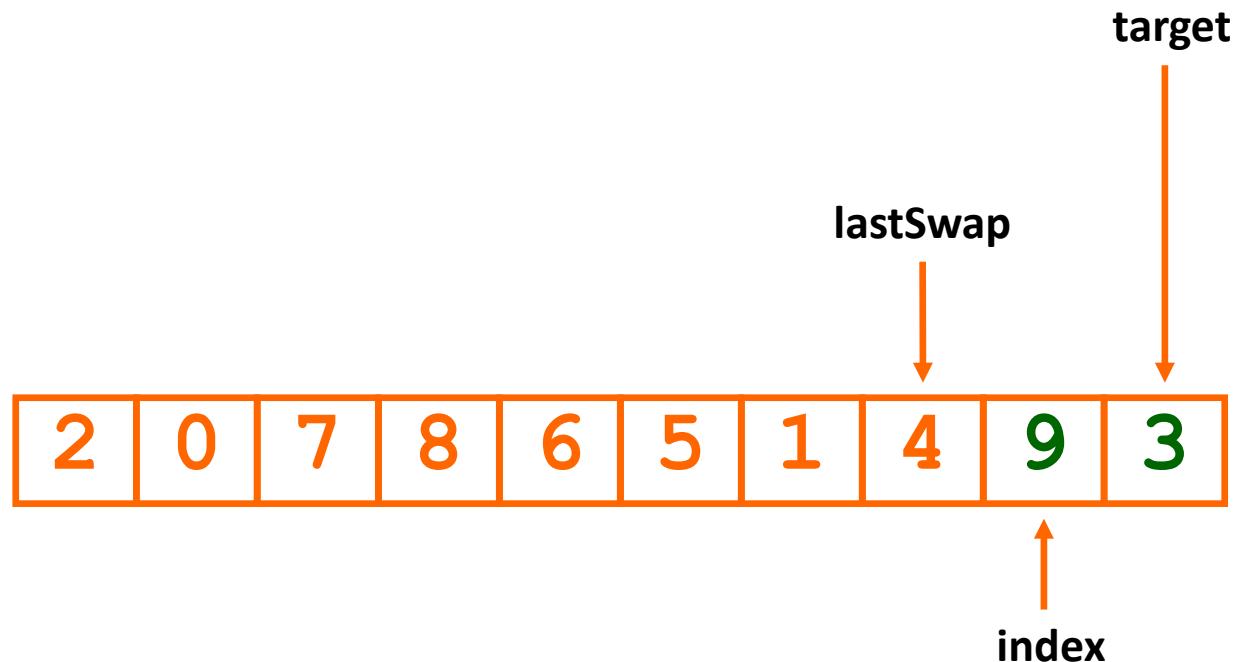
Bubble Sort Animation



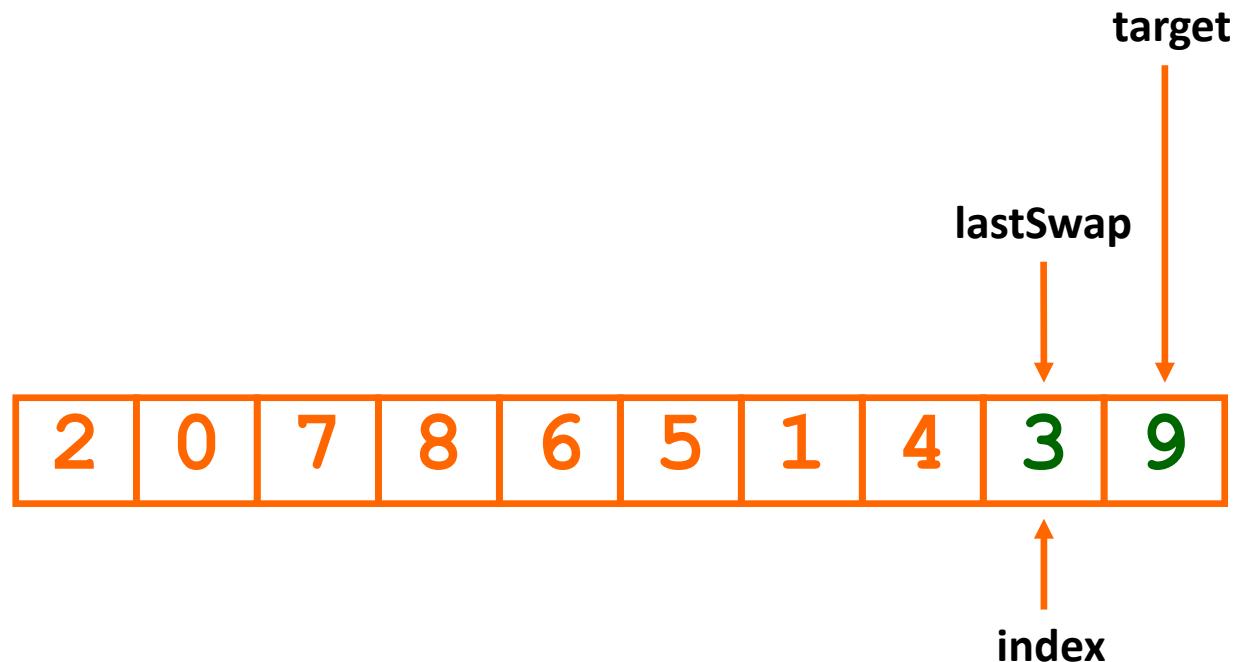
Bubble Sort Animation



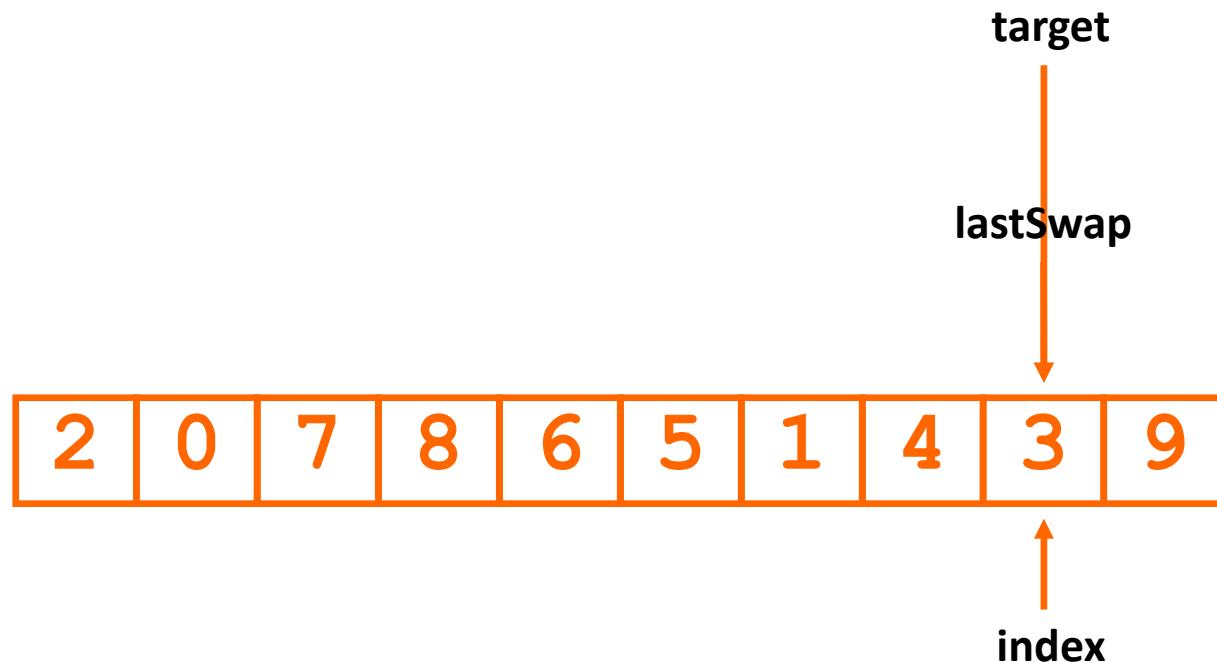
Bubble Sort Animation



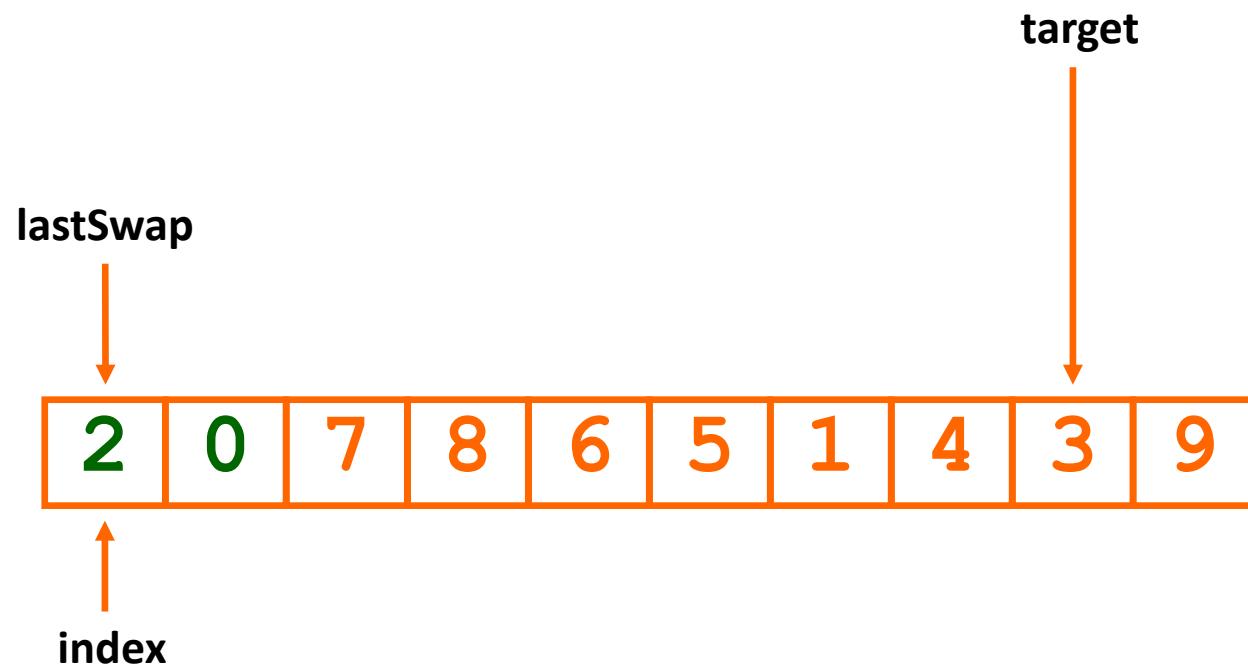
Bubble Sort Animation



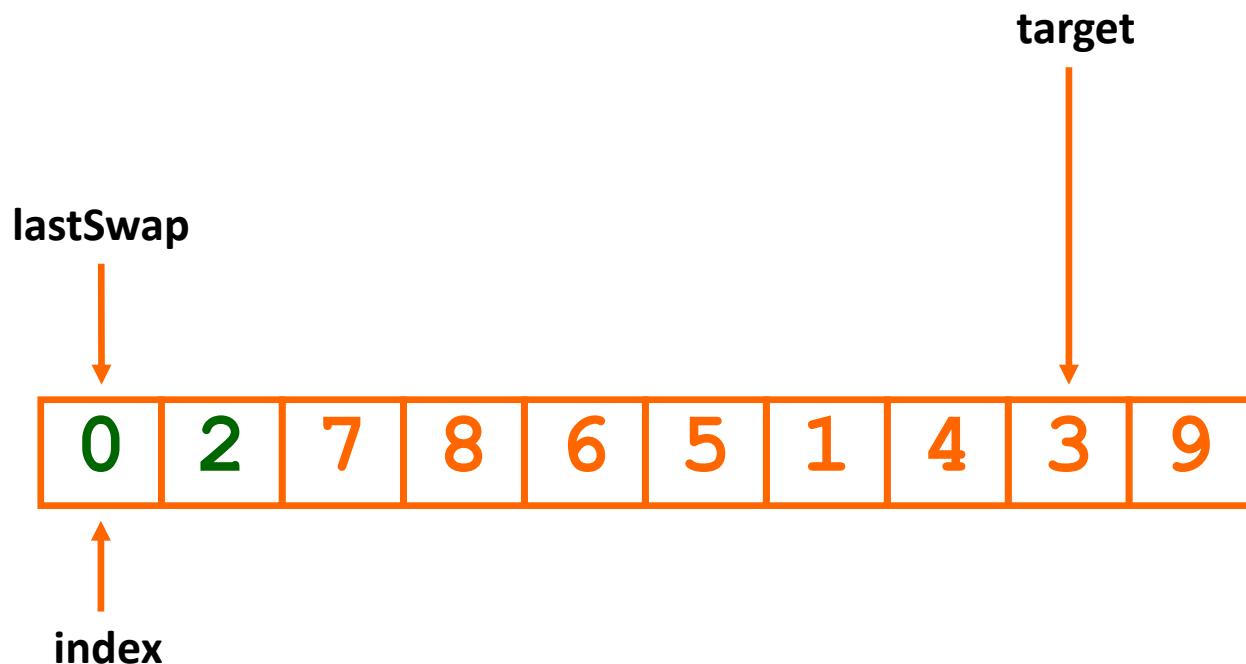
Bubble Sort Animation



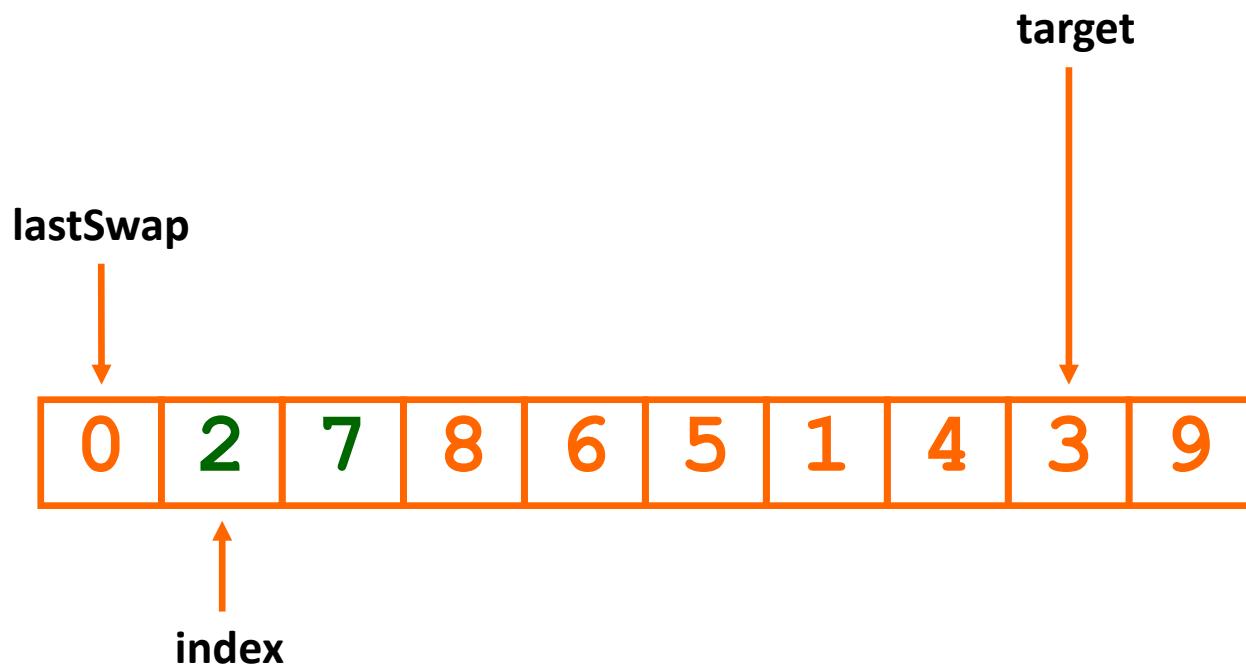
Bubble Sort Animation



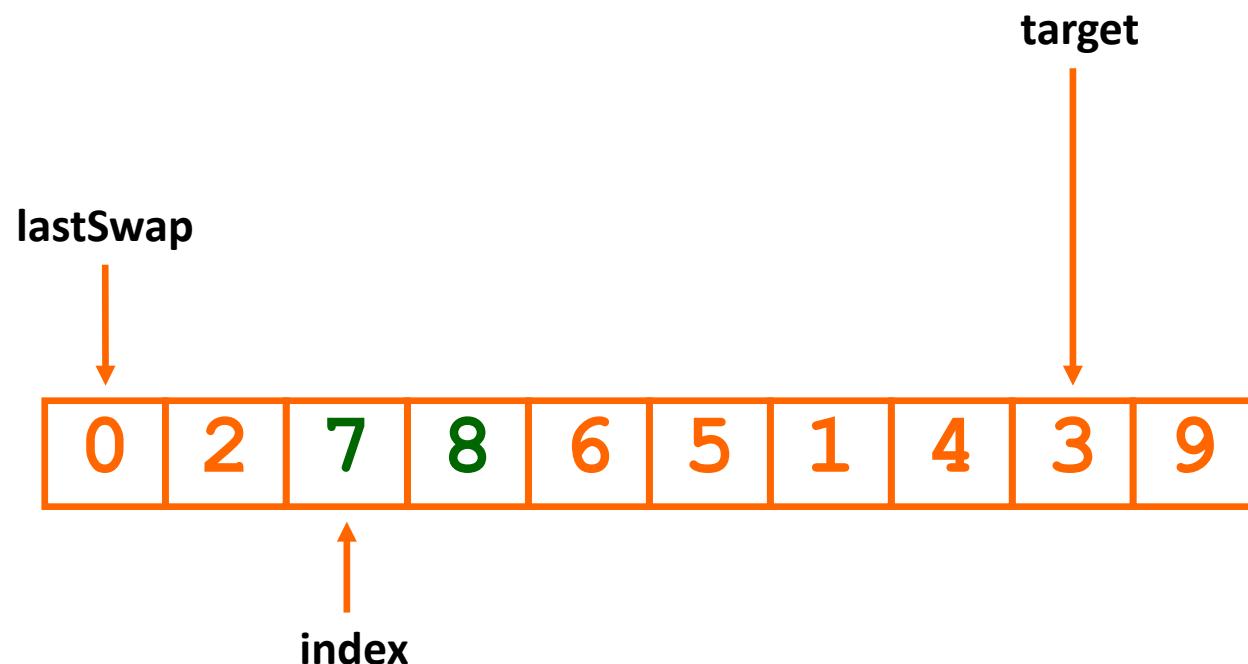
Bubble Sort Animation



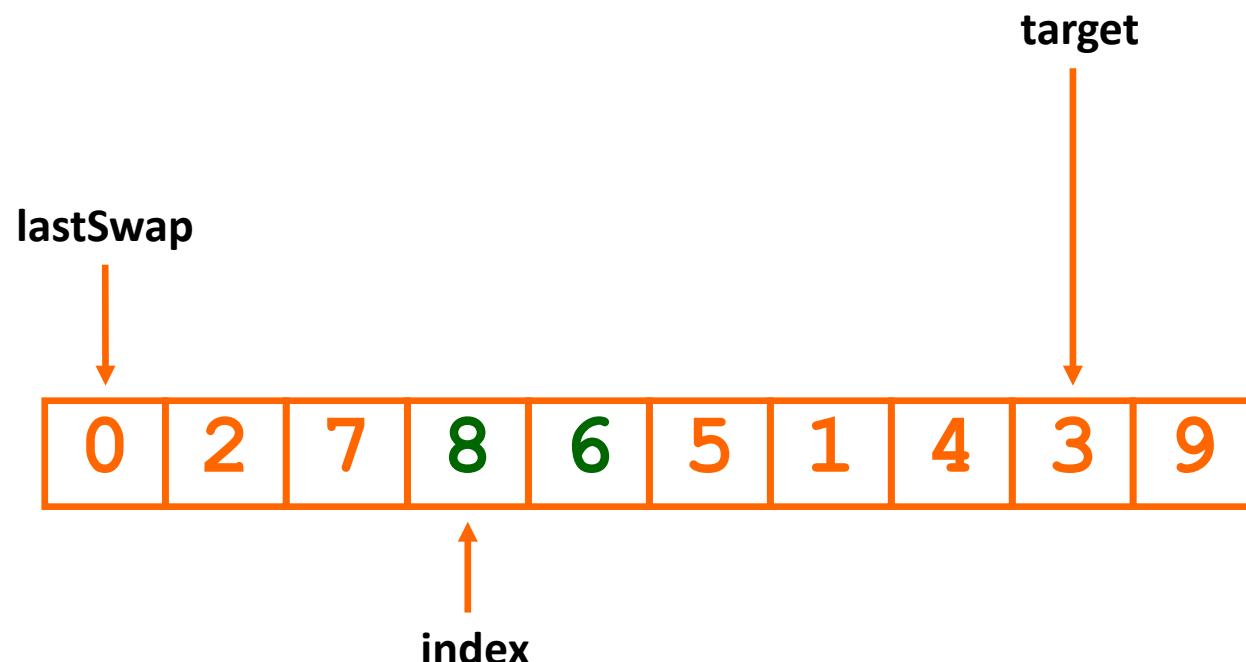
Bubble Sort Animation



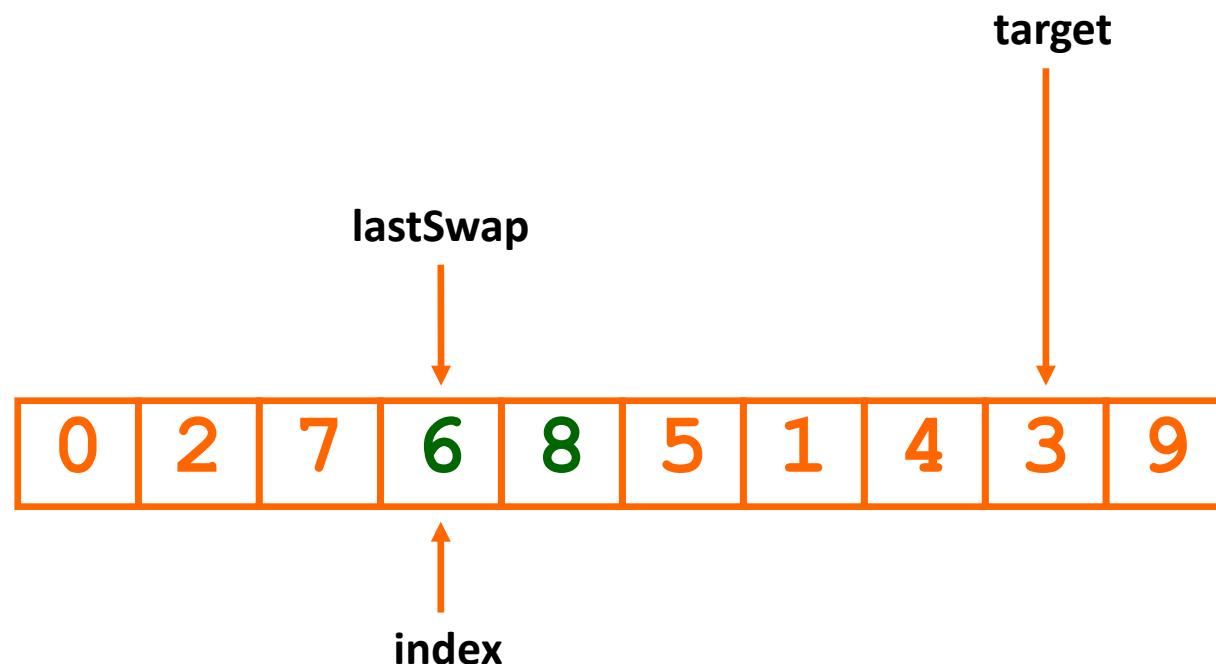
Bubble Sort Animation



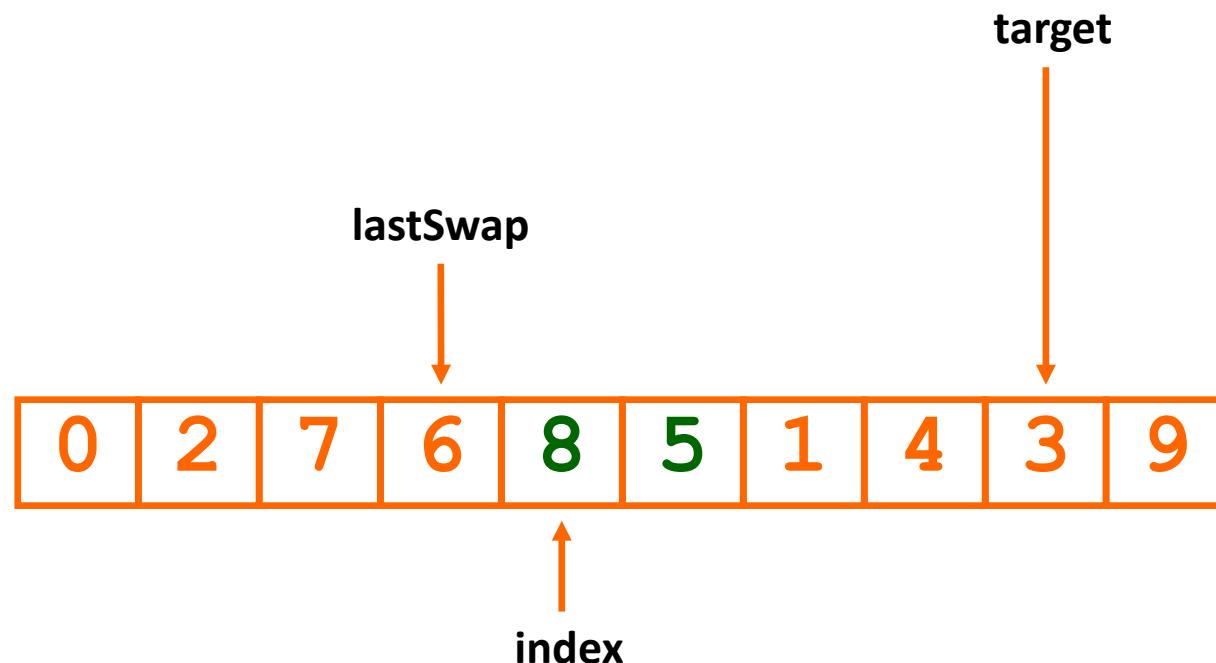
Bubble Sort Animation



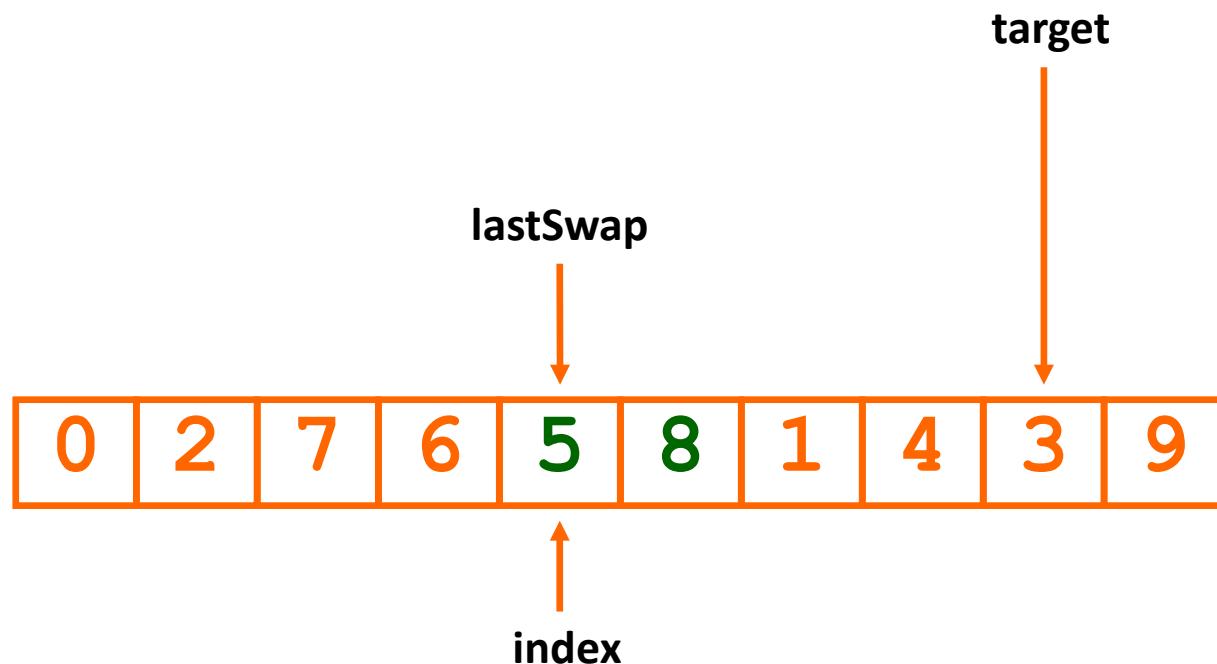
Bubble Sort Animation



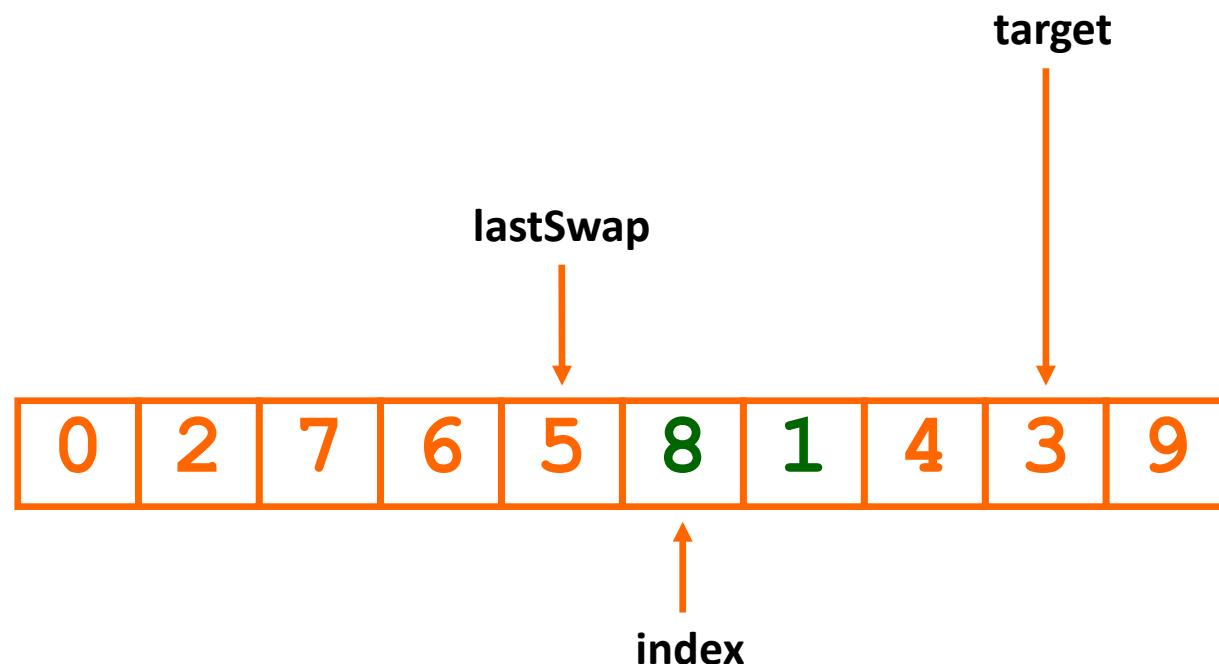
Bubble Sort Animation



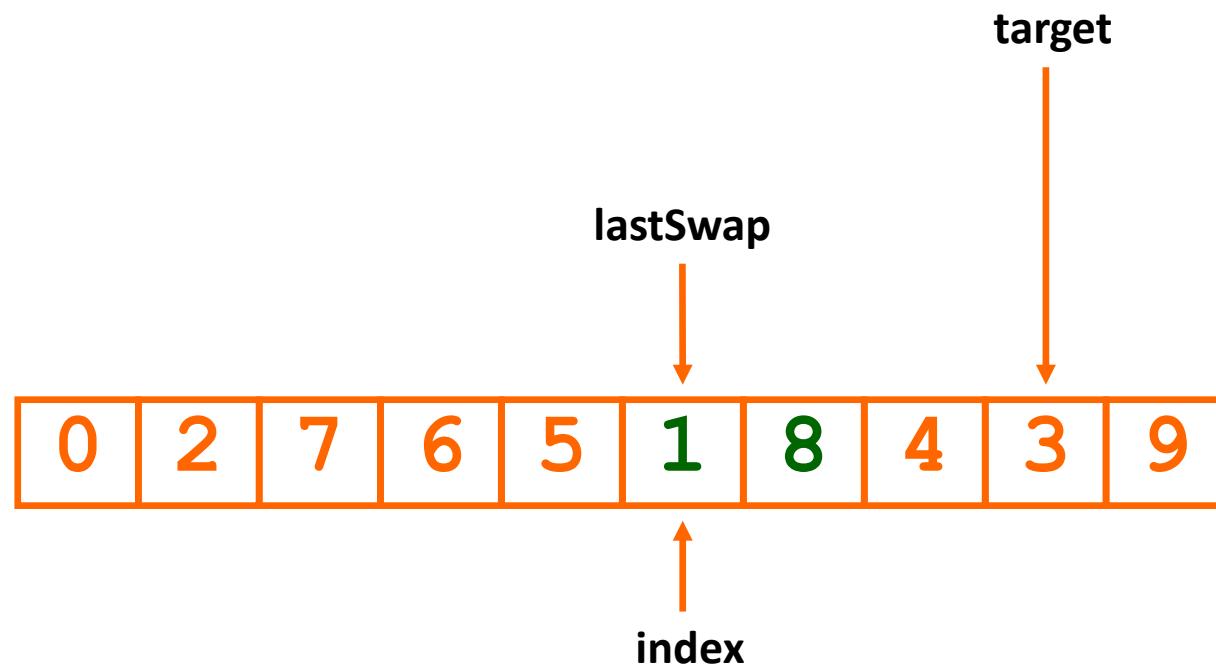
Bubble Sort Animation



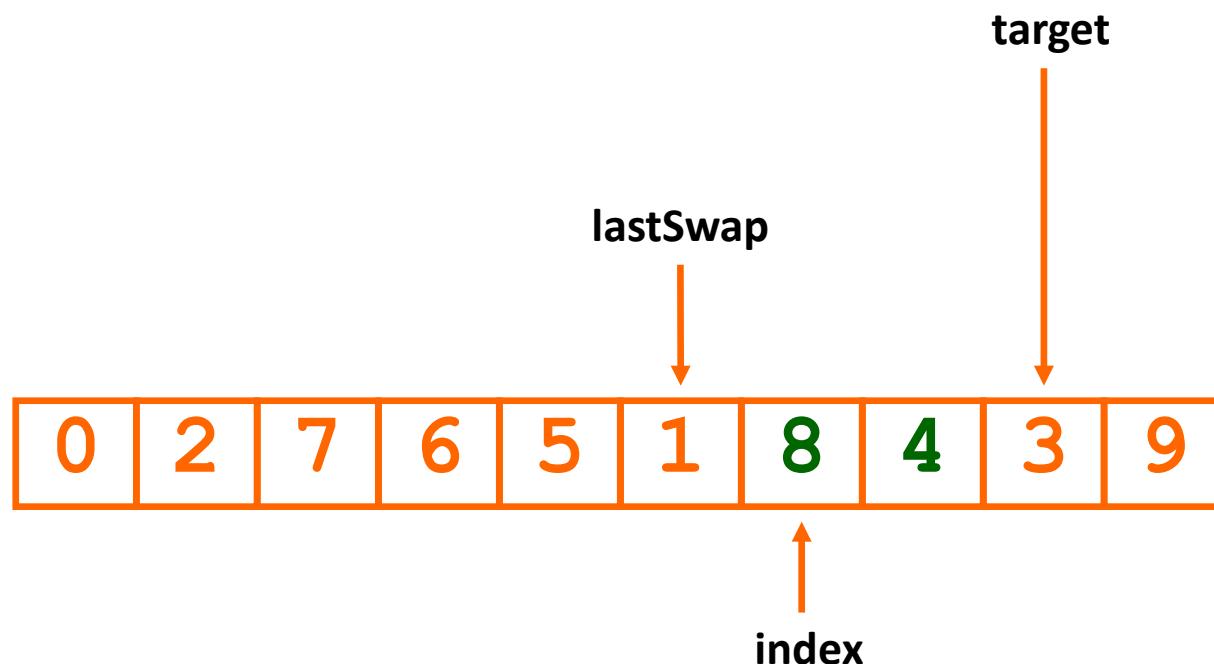
Bubble Sort Animation



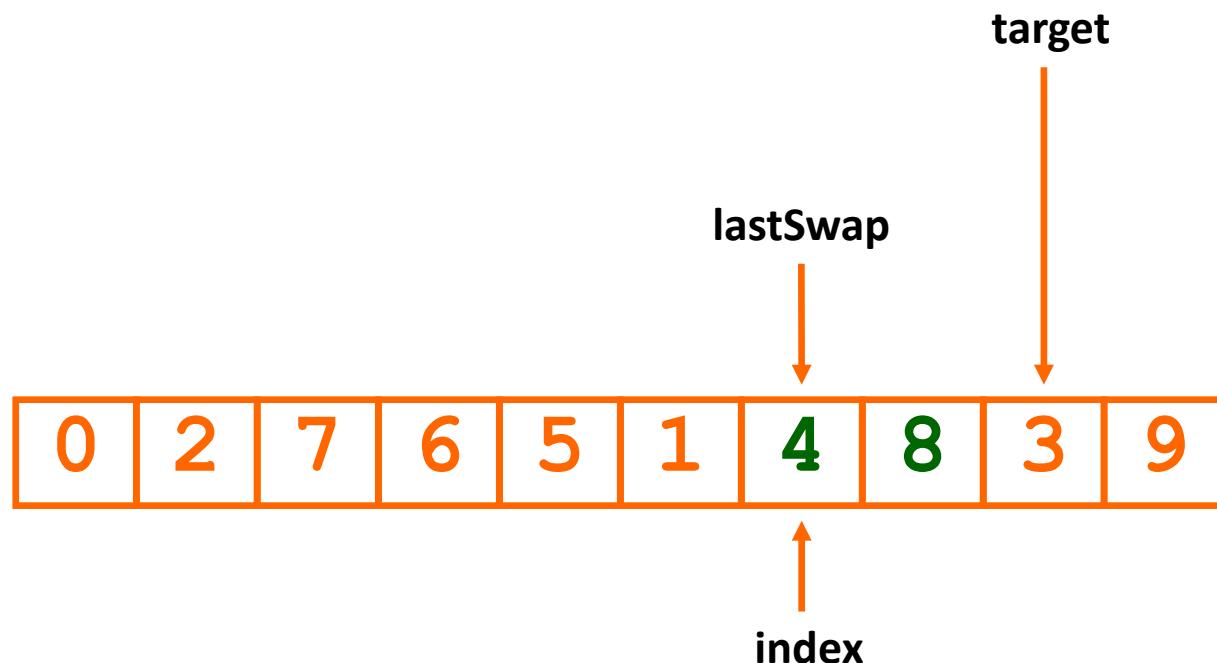
Bubble Sort Animation



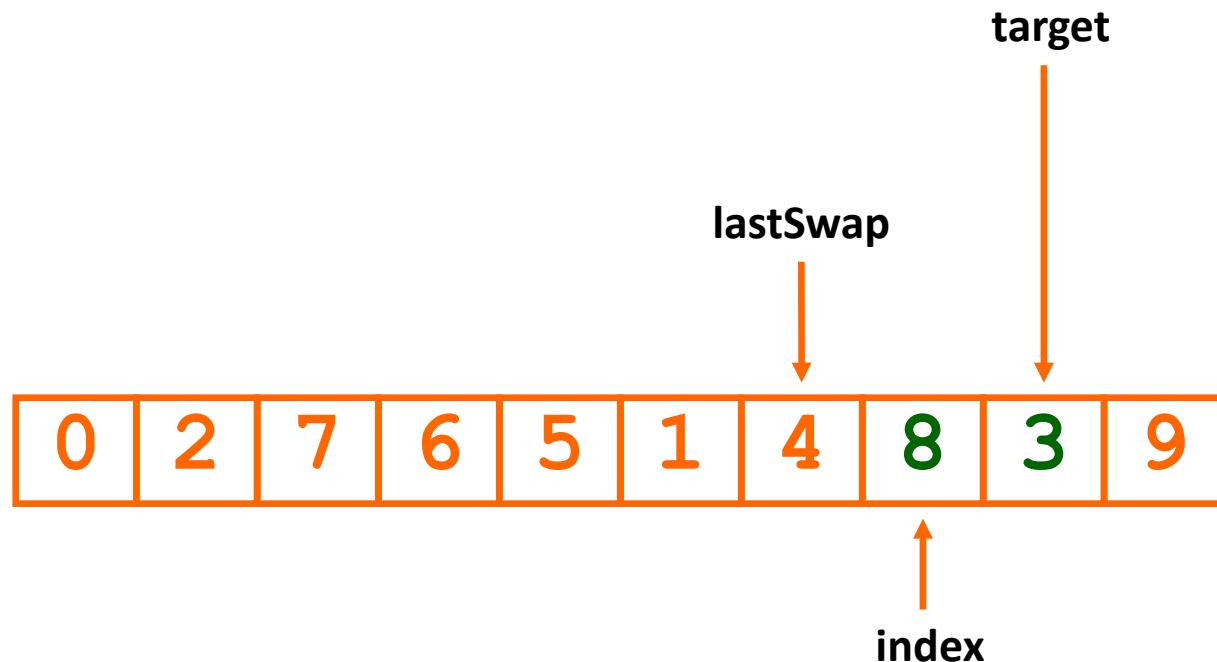
Bubble Sort Animation



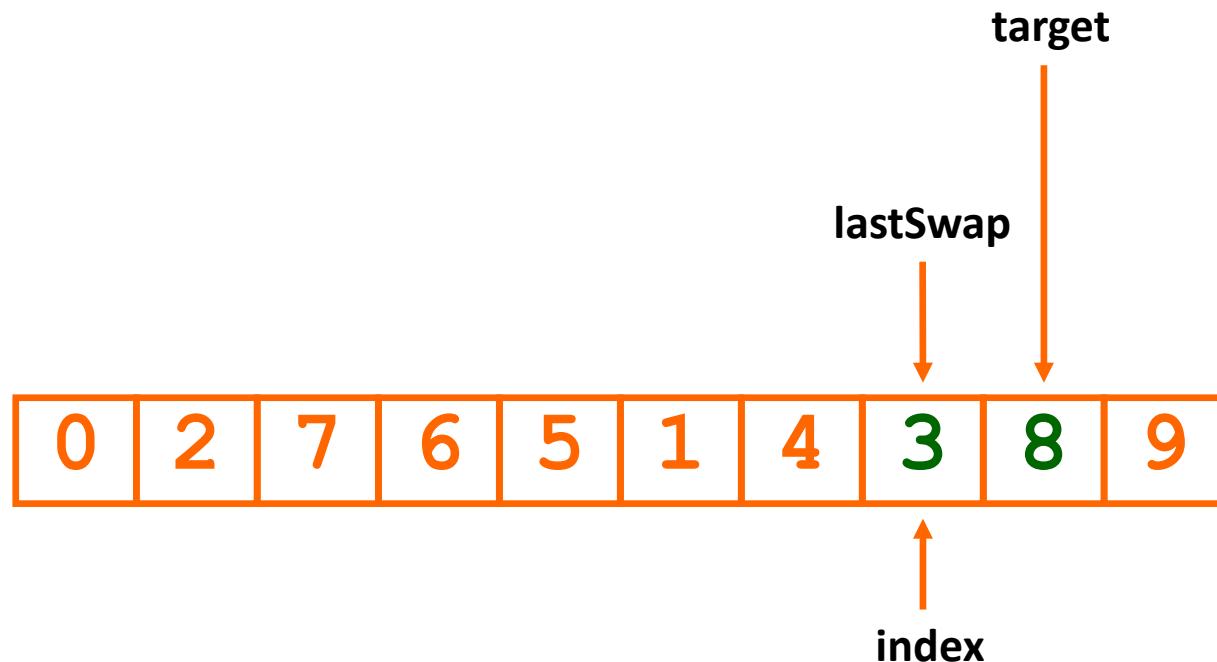
Bubble Sort Animation



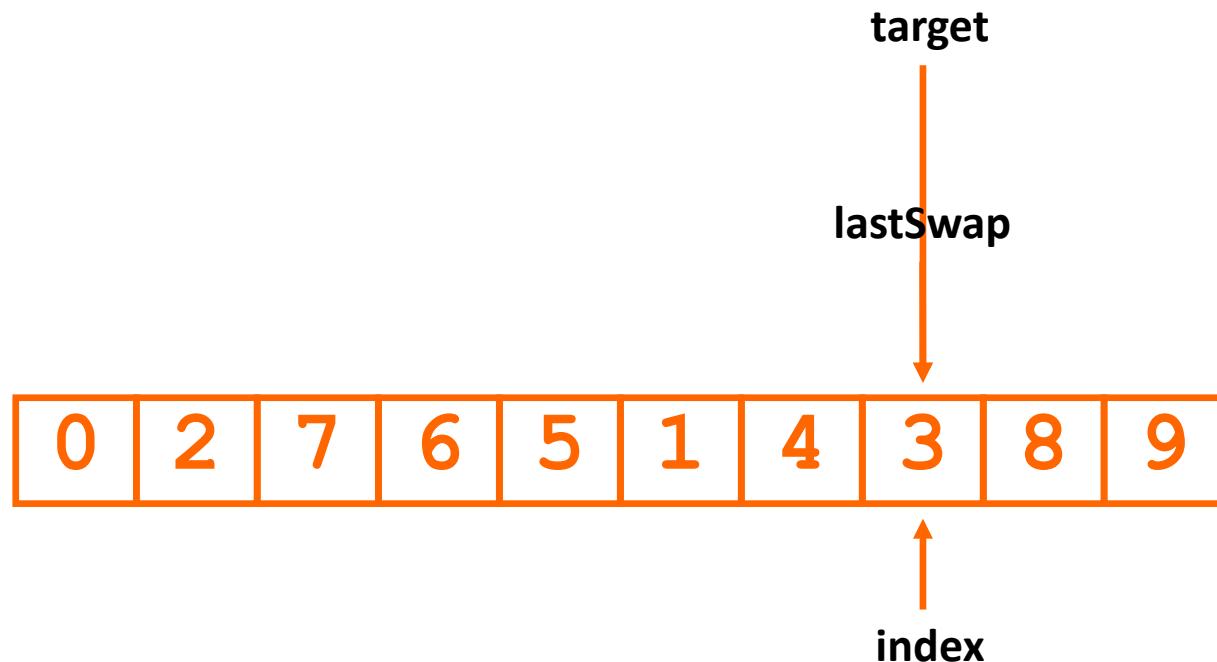
Bubble Sort Animation



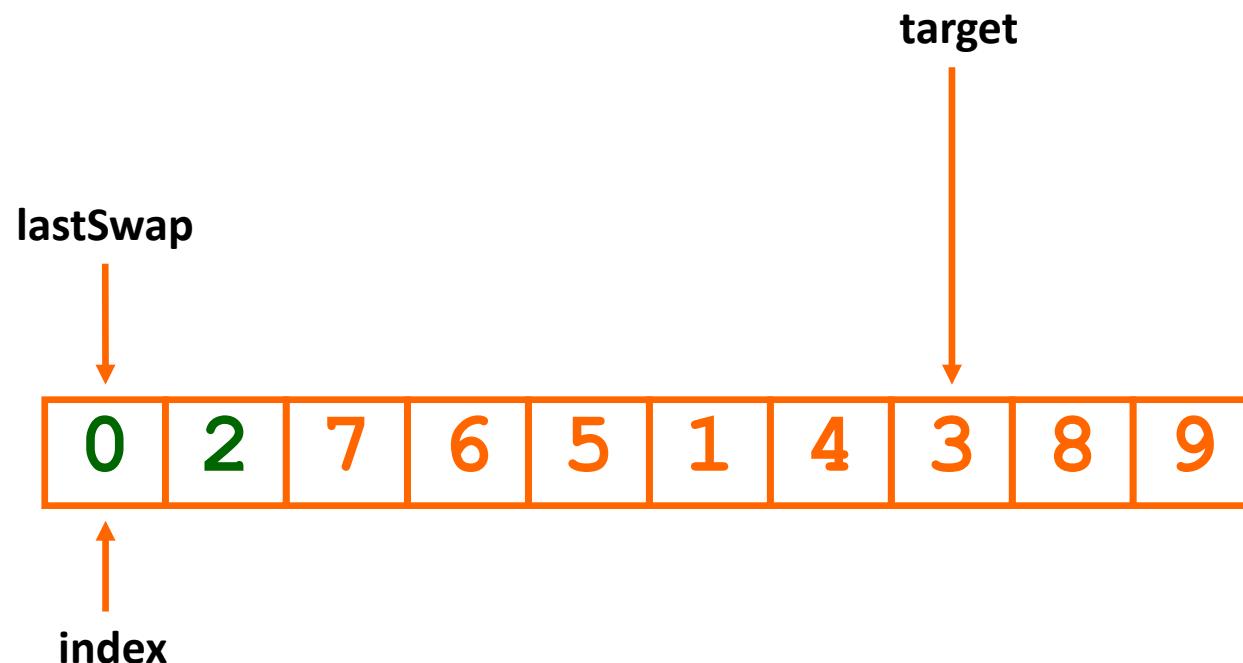
Bubble Sort Animation



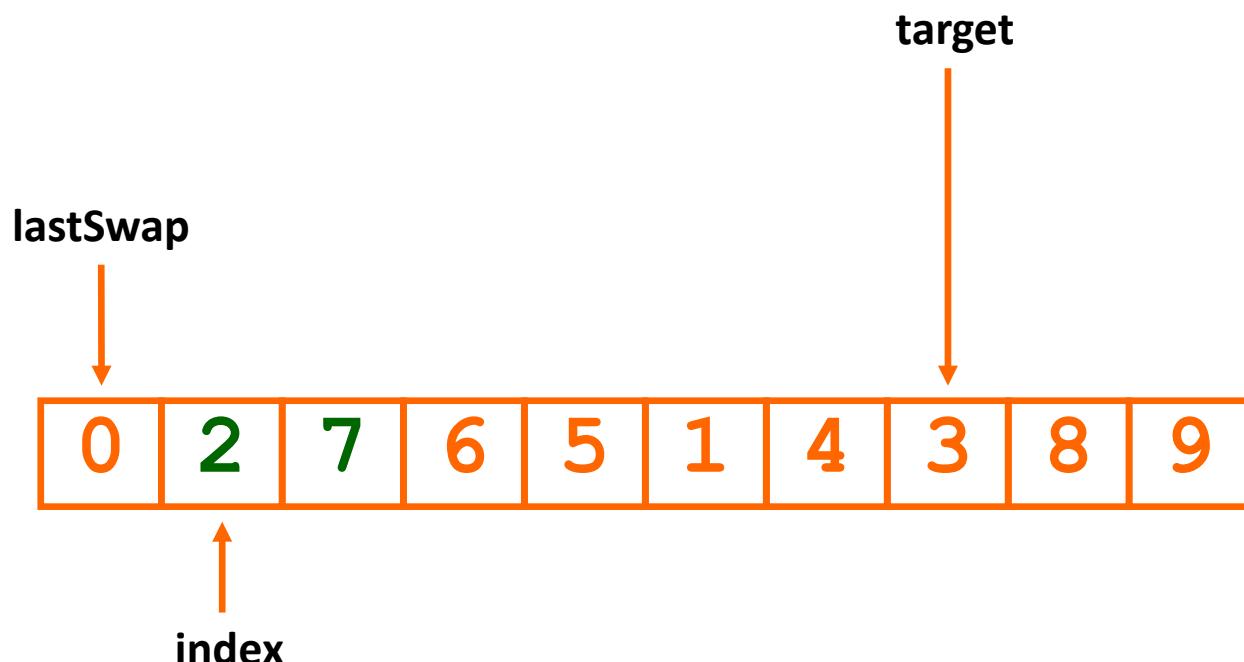
Bubble Sort Animation



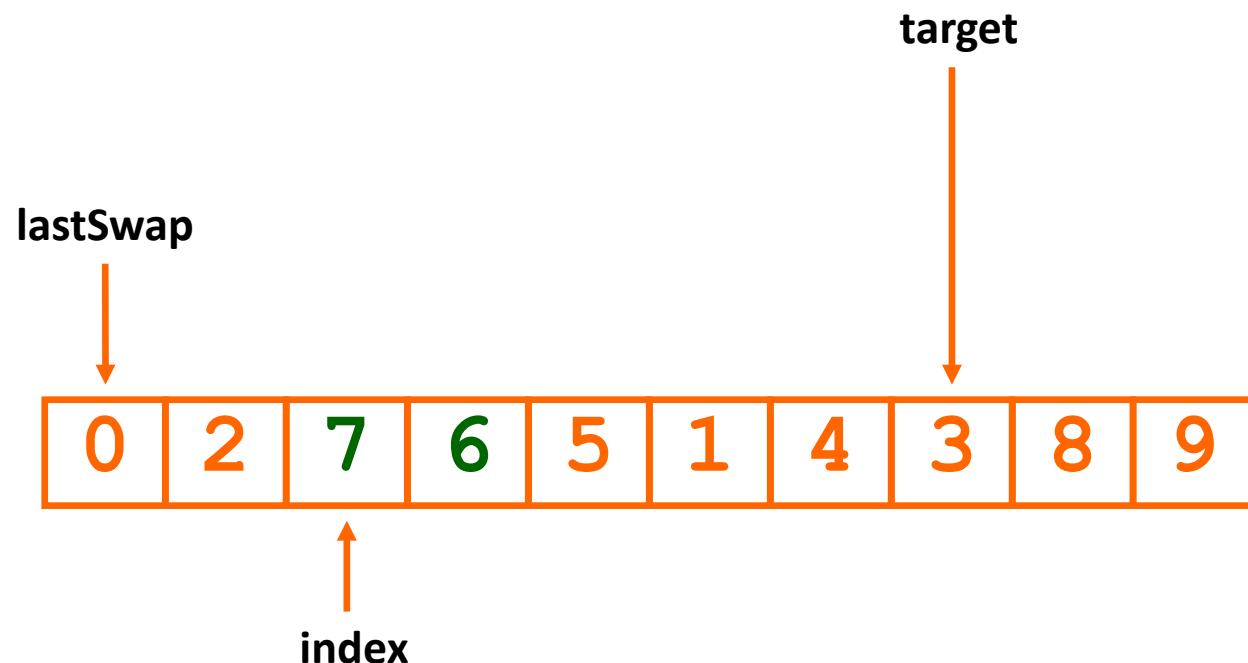
Bubble Sort Animation



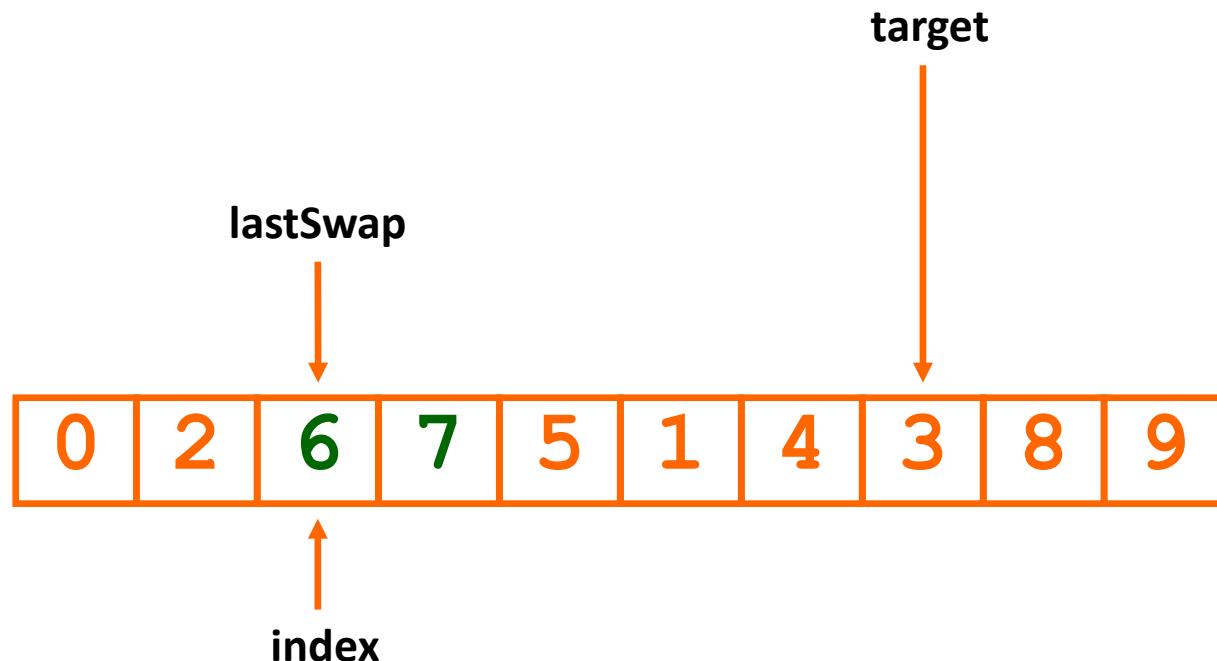
Bubble Sort Animation



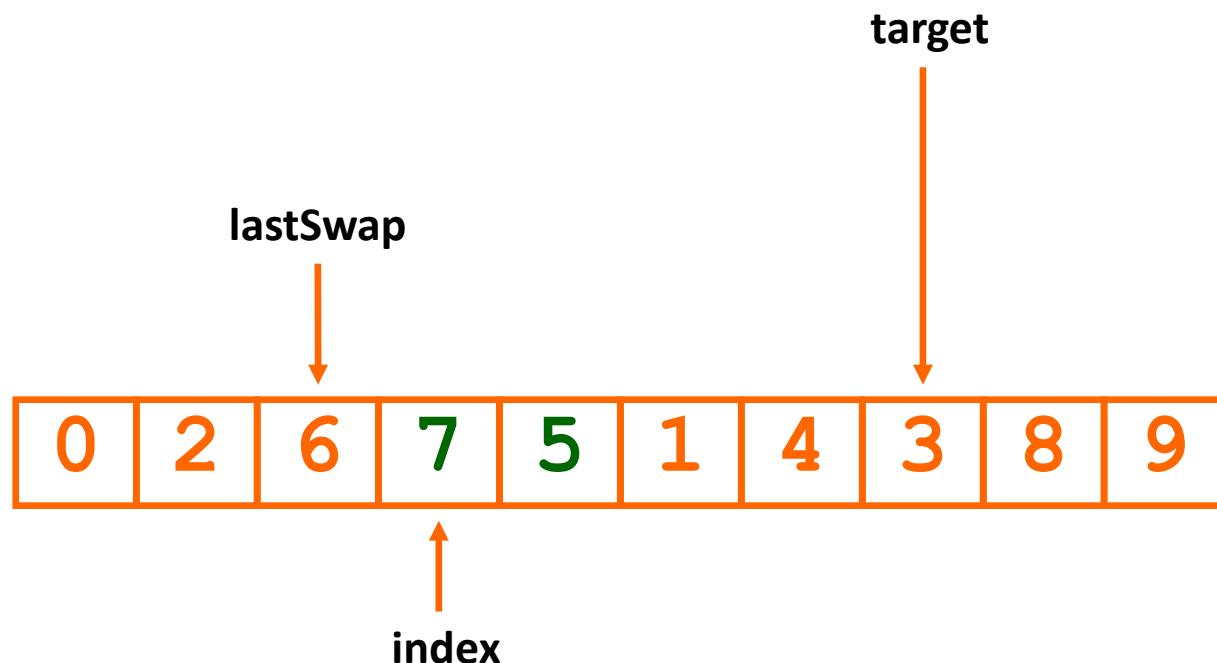
Bubble Sort Animation



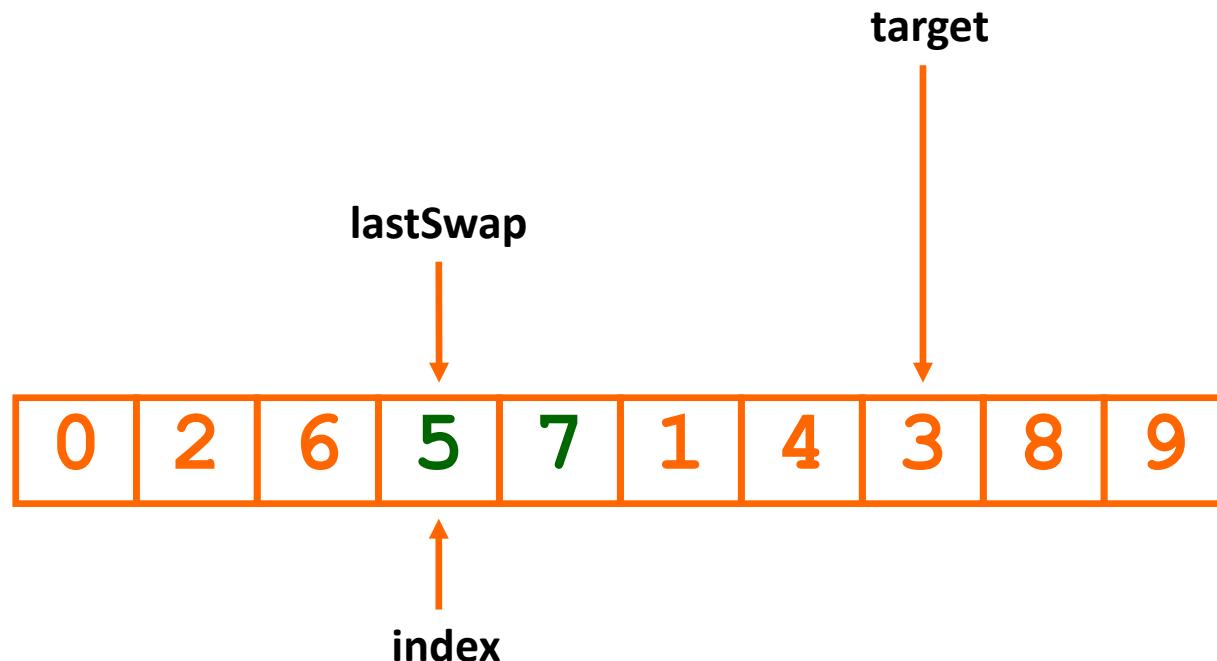
Bubble Sort Animation



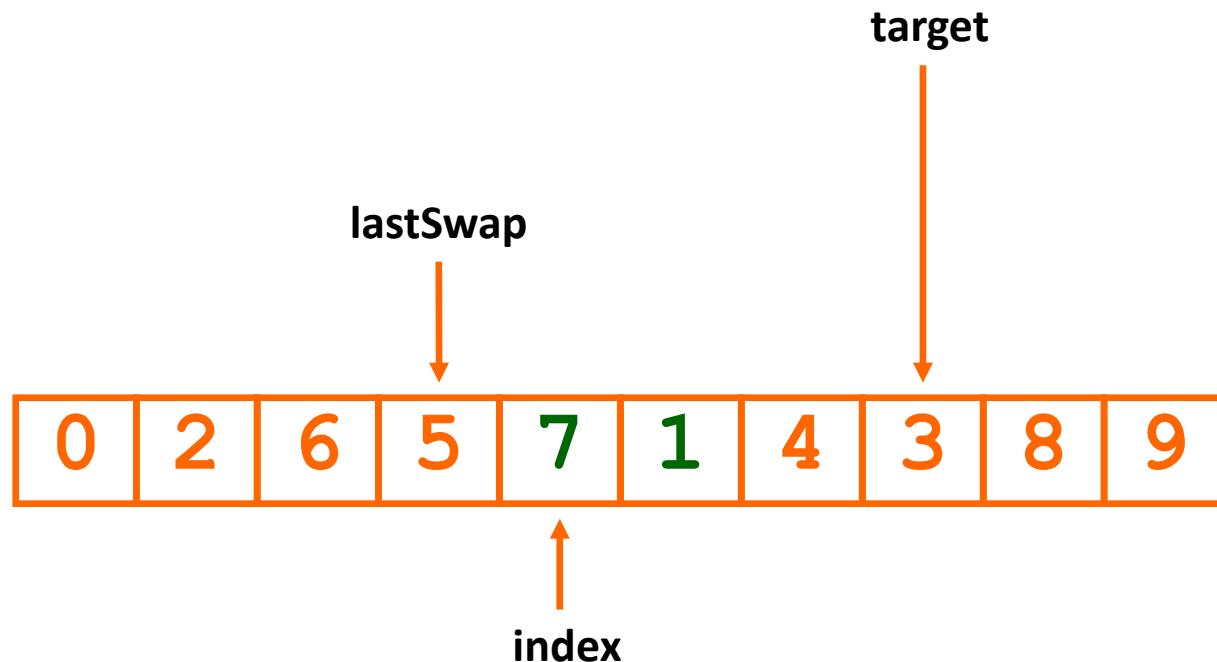
Bubble Sort Animation



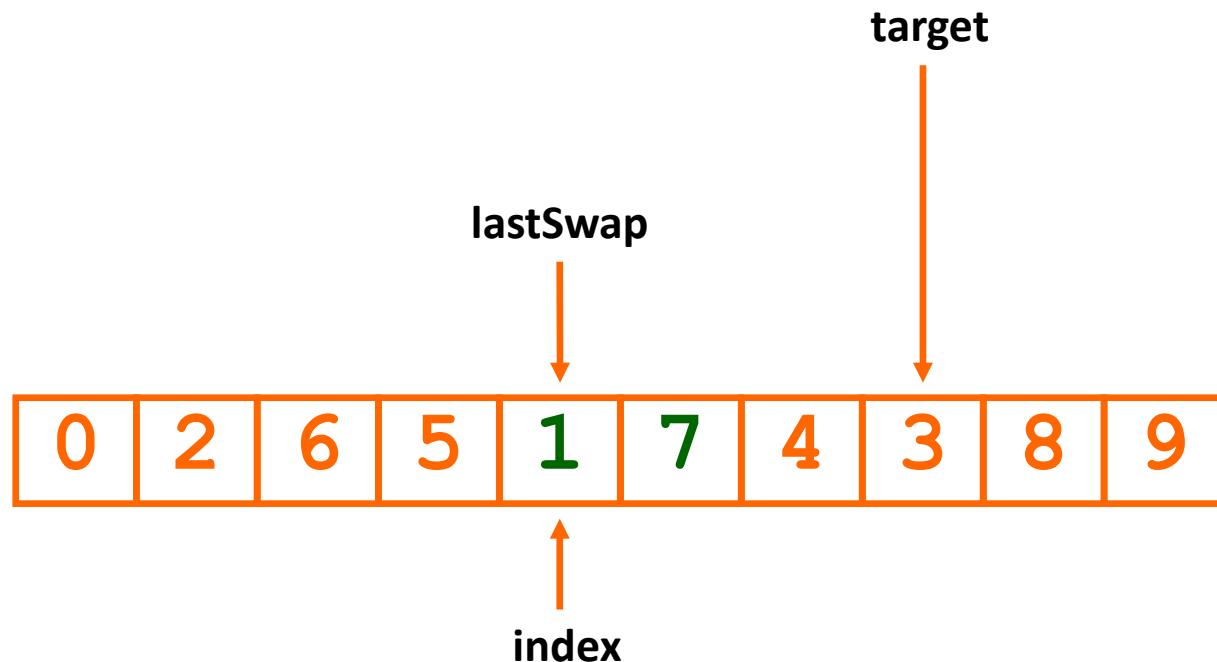
Bubble Sort Animation



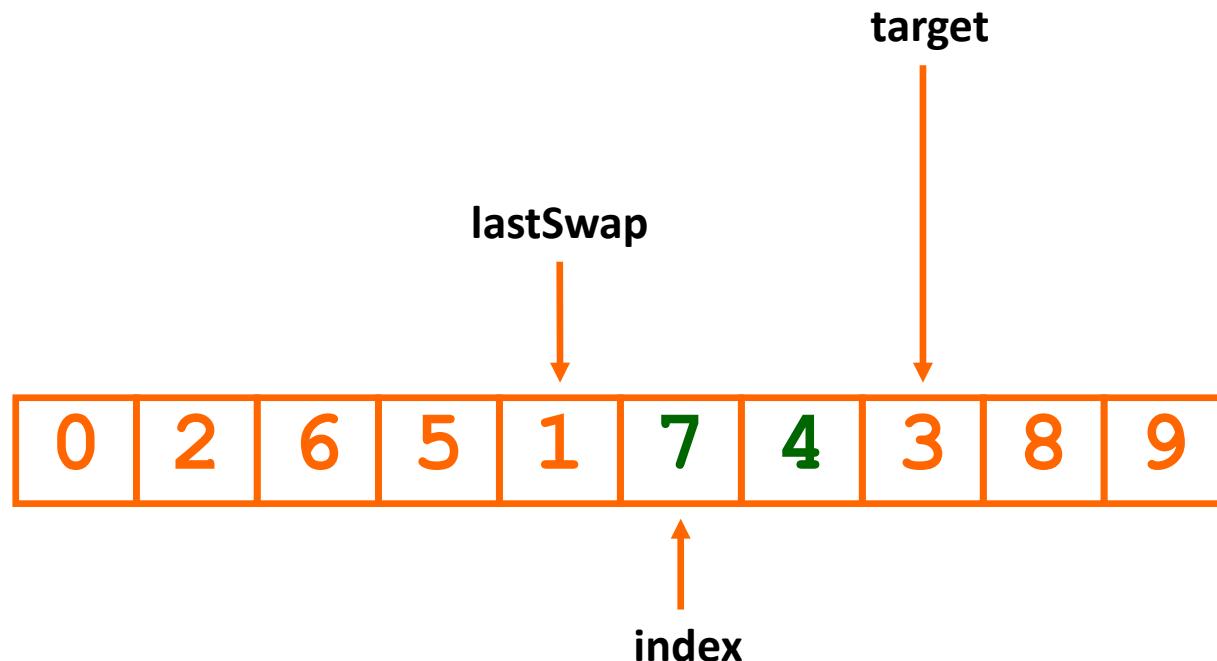
Bubble Sort Animation



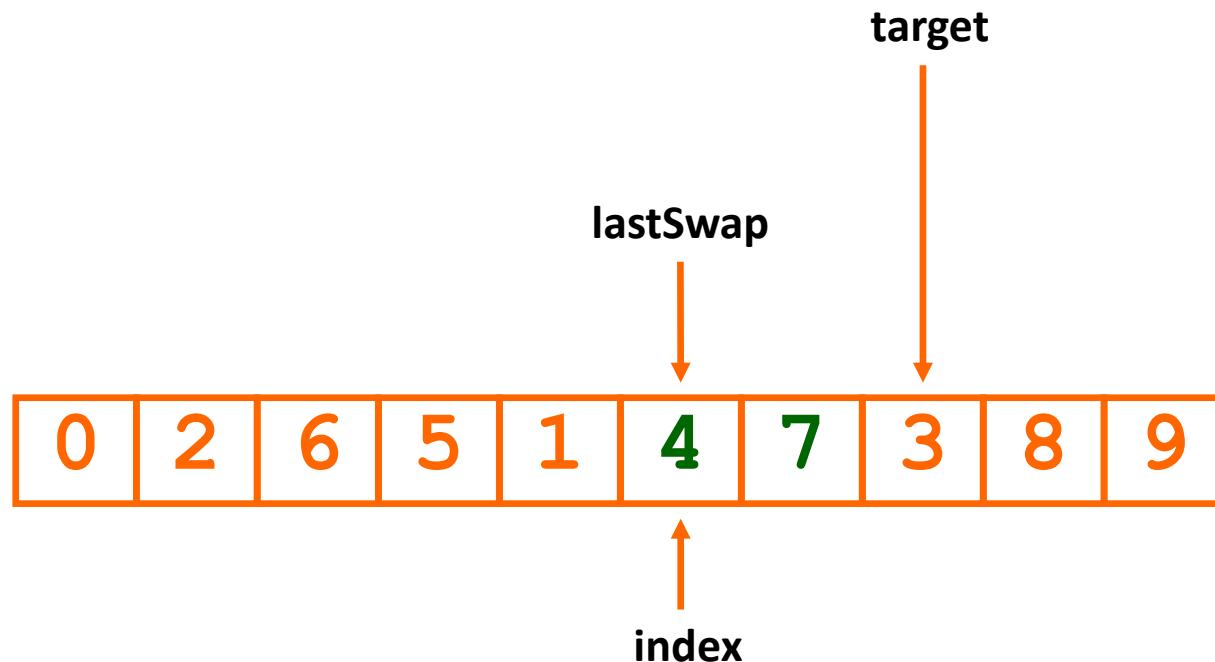
Bubble Sort Animation



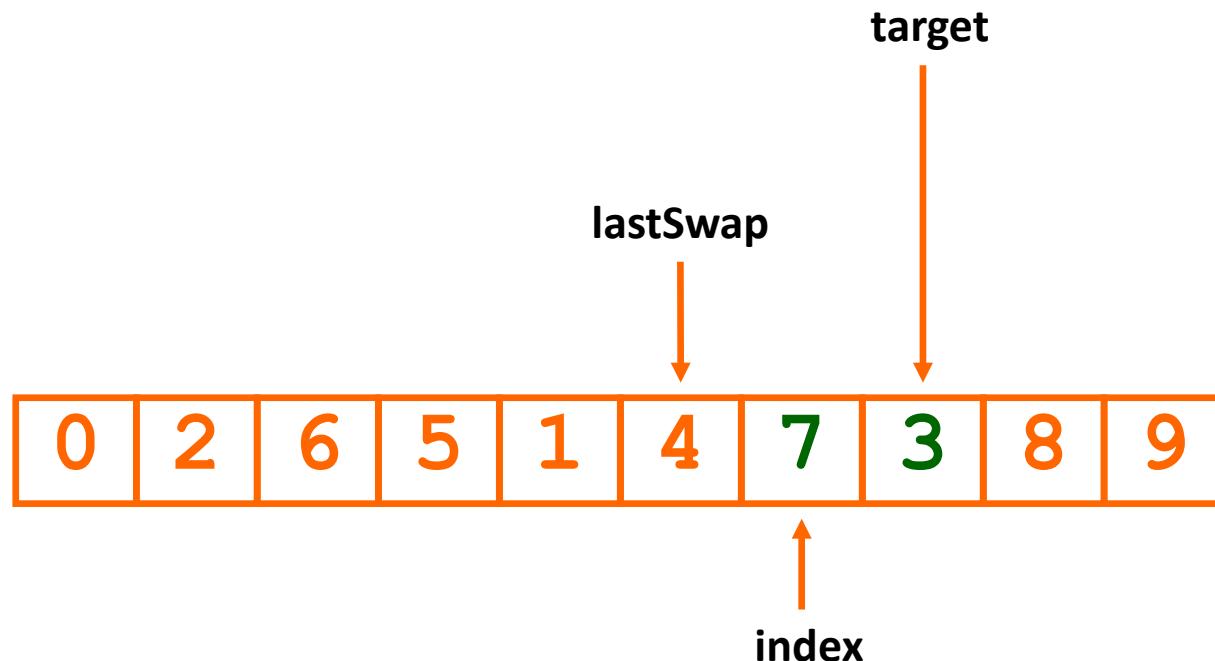
Bubble Sort Animation



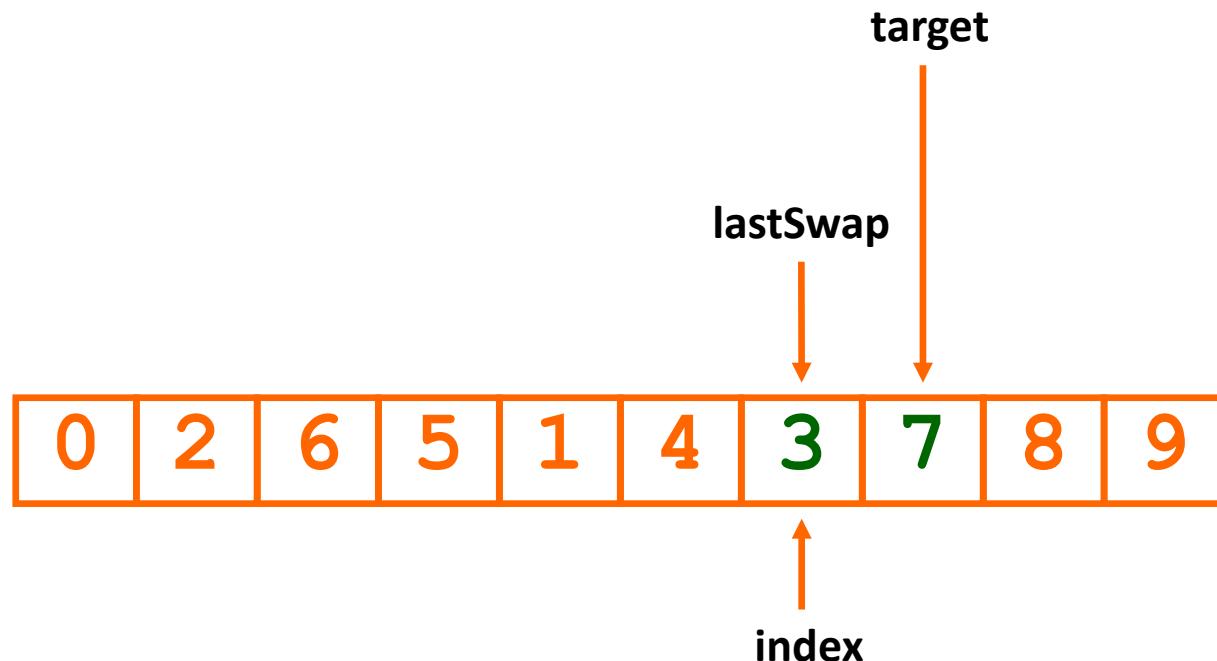
Bubble Sort Animation



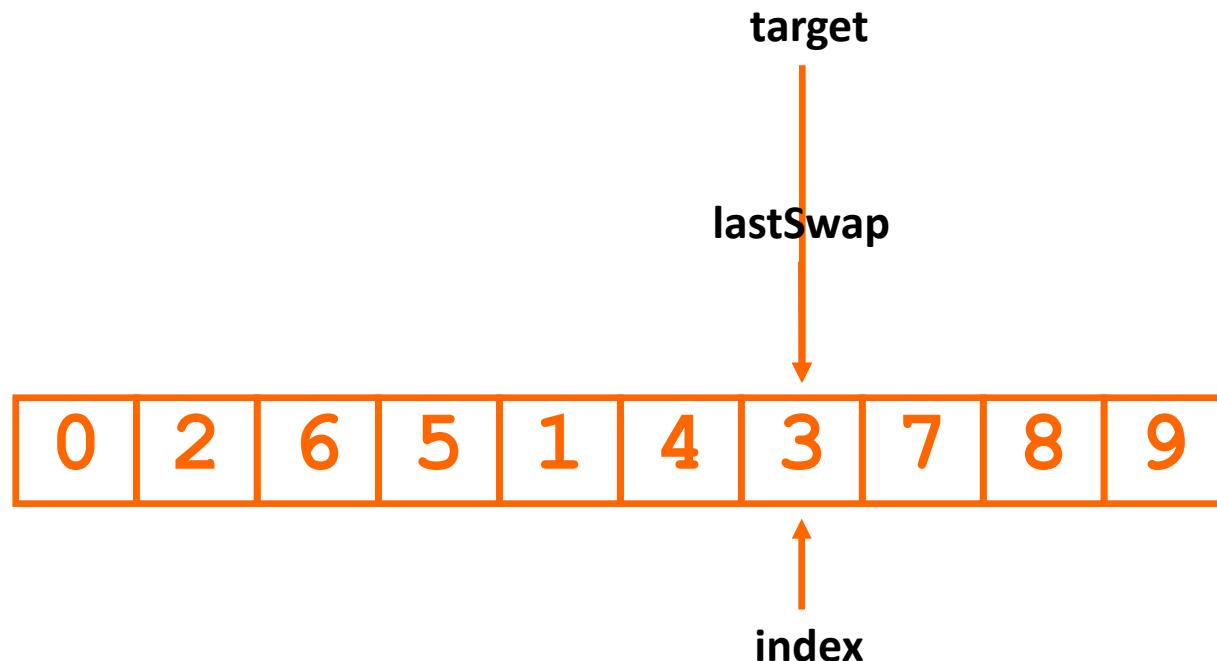
Bubble Sort Animation



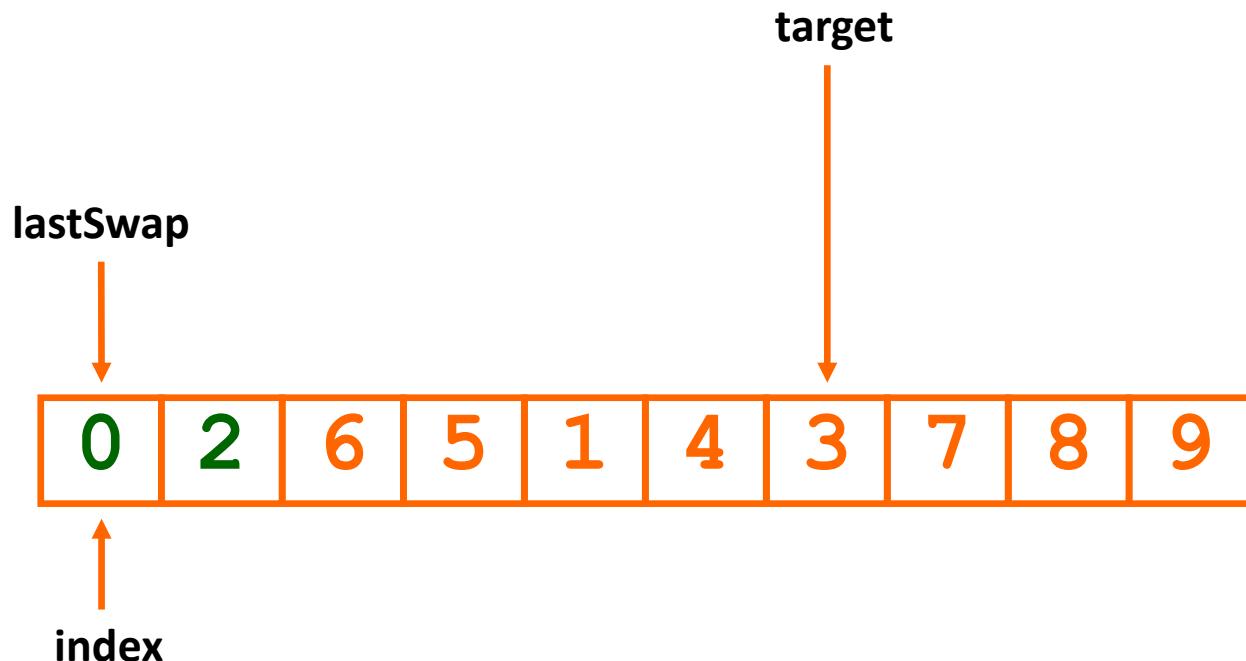
Bubble Sort Animation



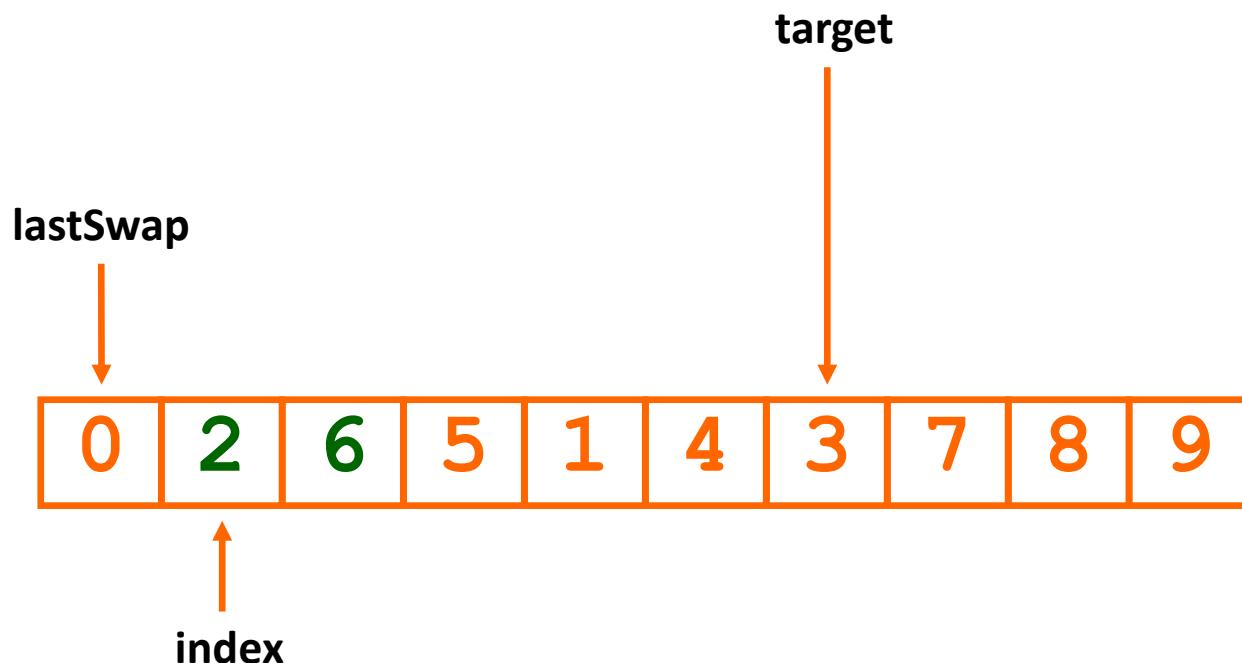
Bubble Sort Animation



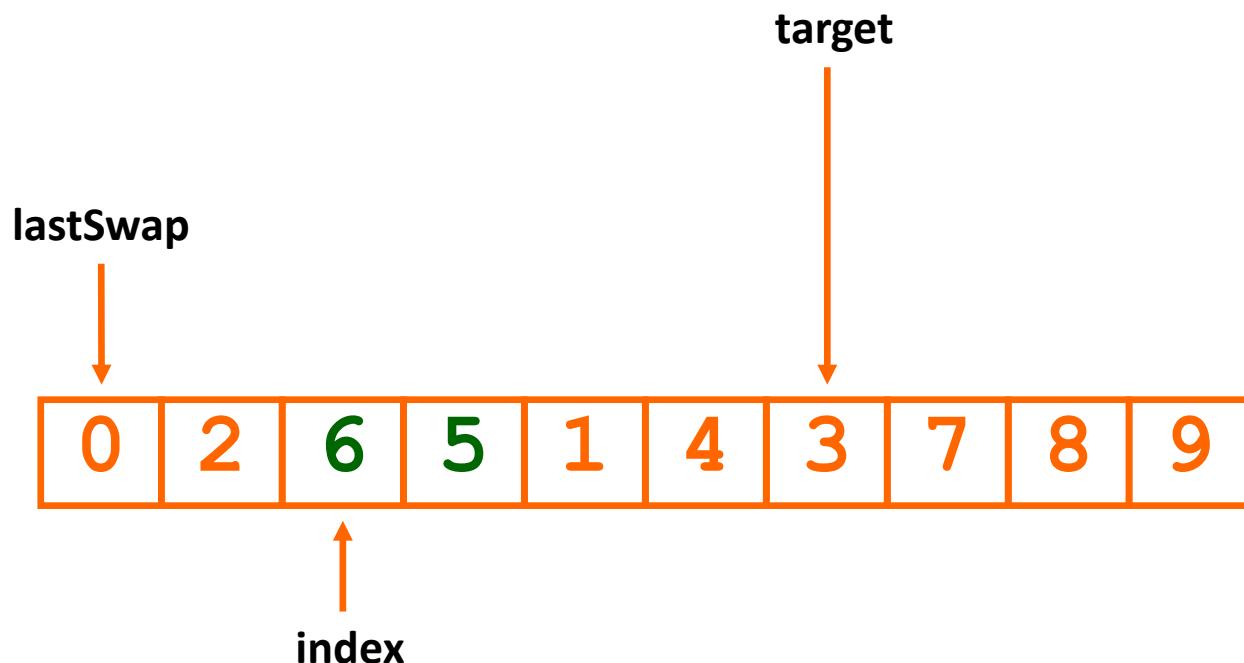
Bubble Sort Animation



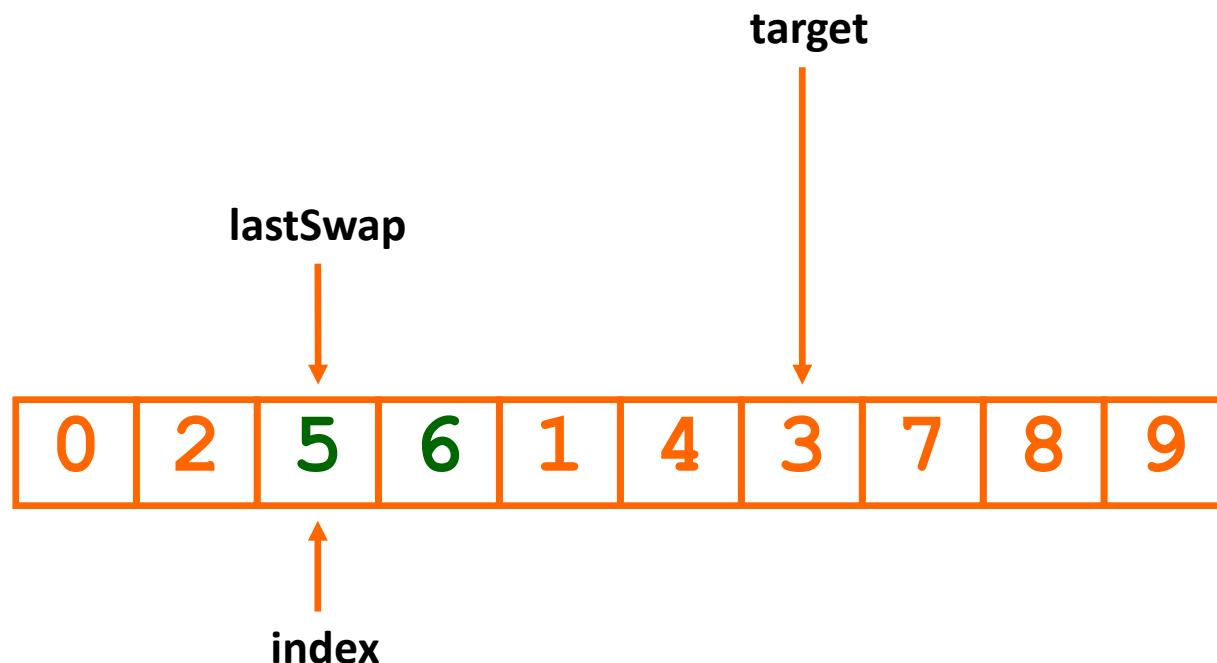
Bubble Sort Animation



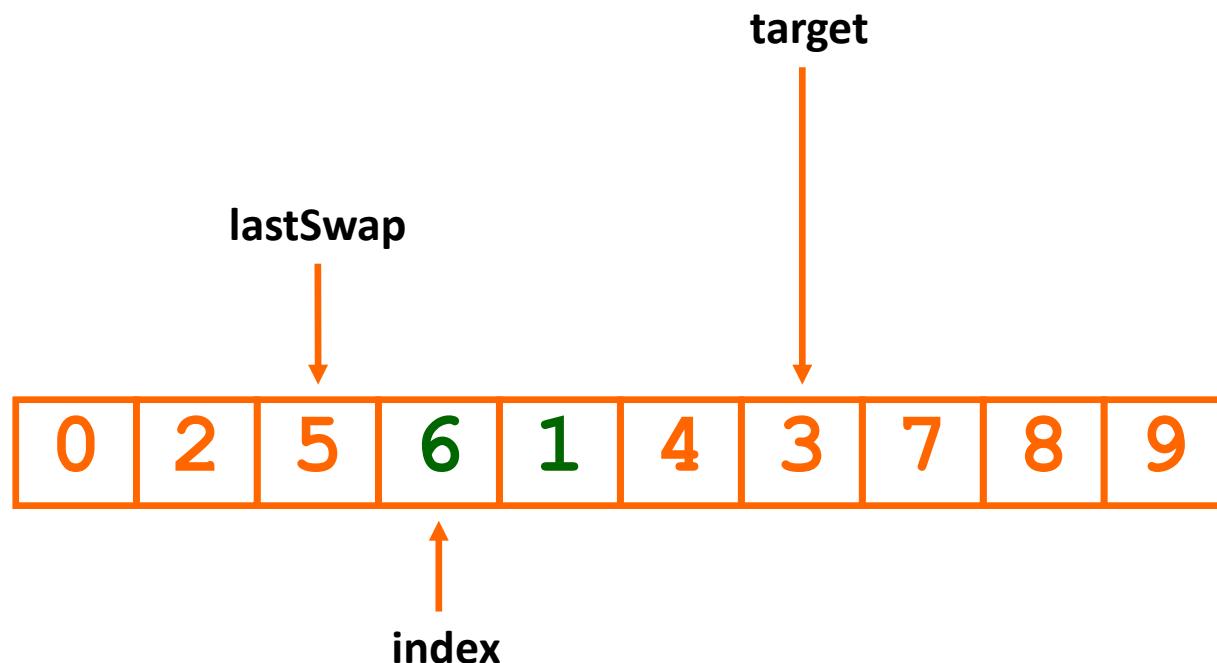
Bubble Sort Animation



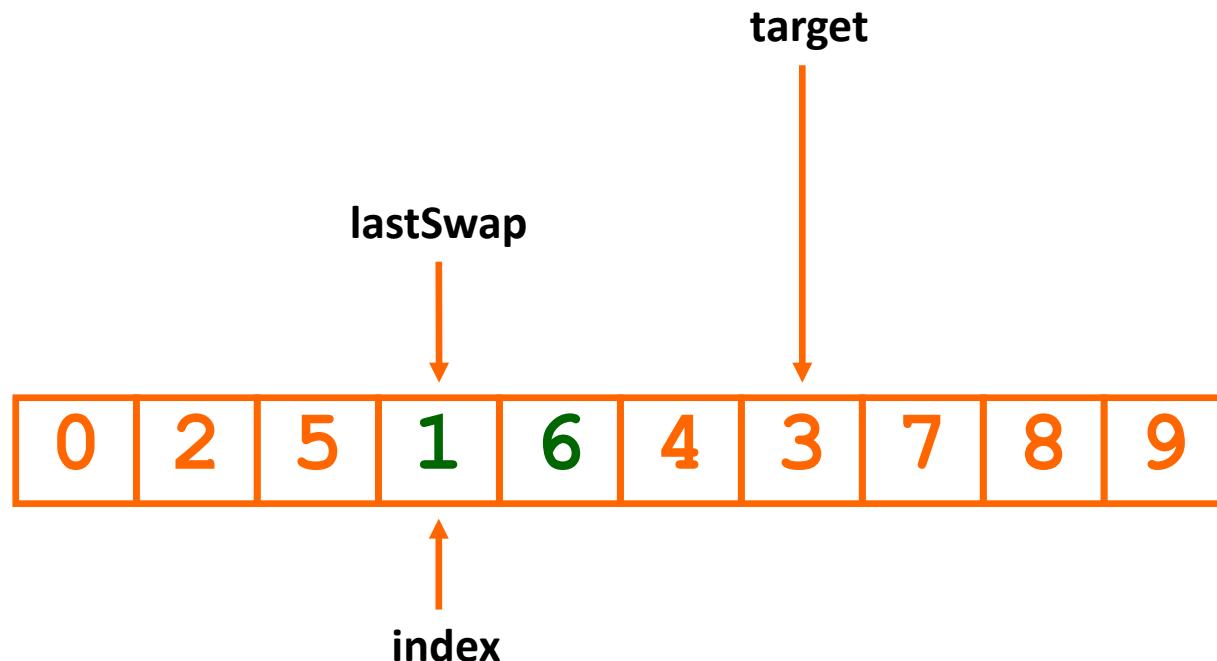
Bubble Sort Animation



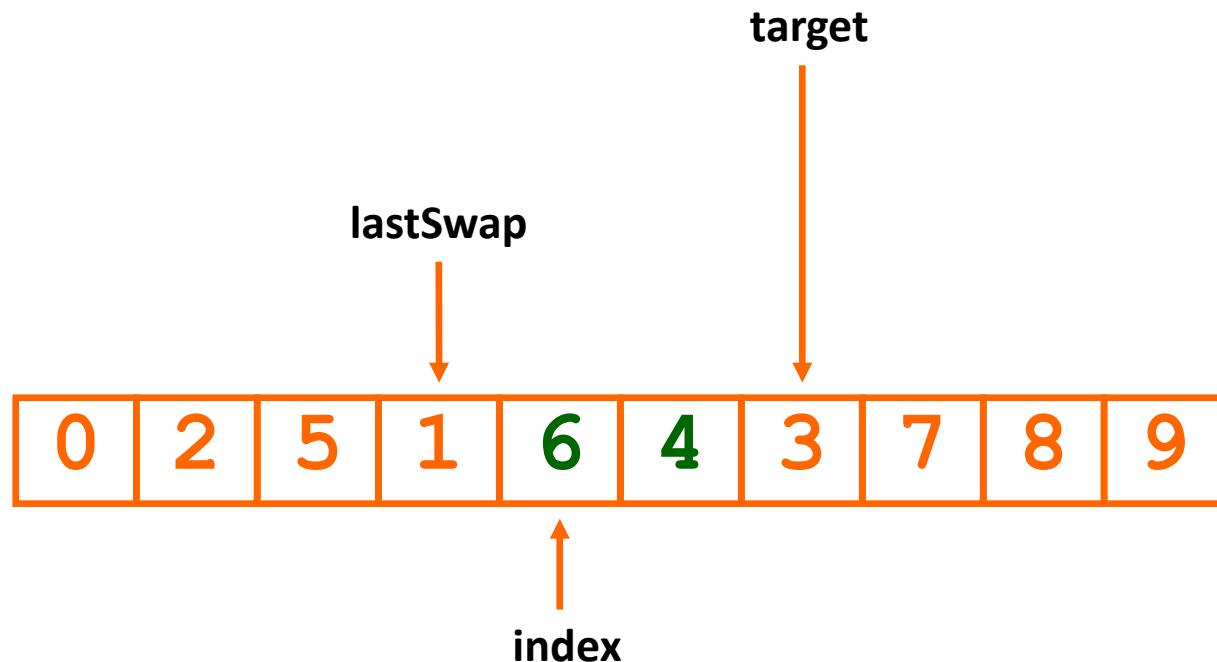
Bubble Sort Animation



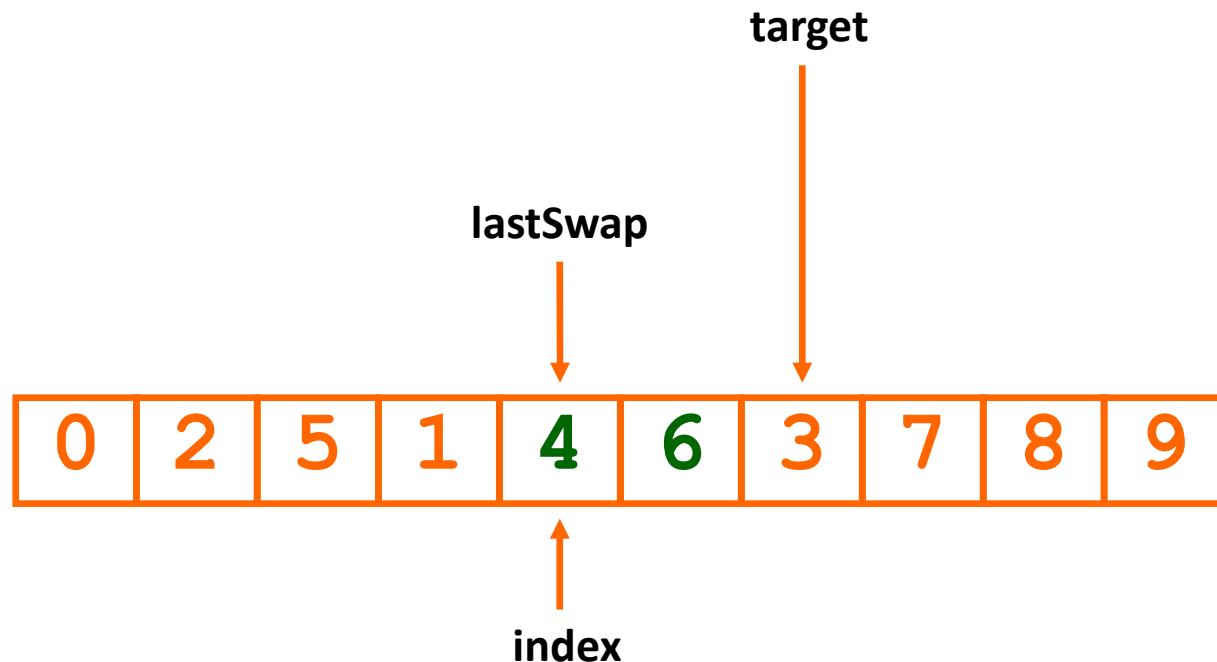
Bubble Sort Animation



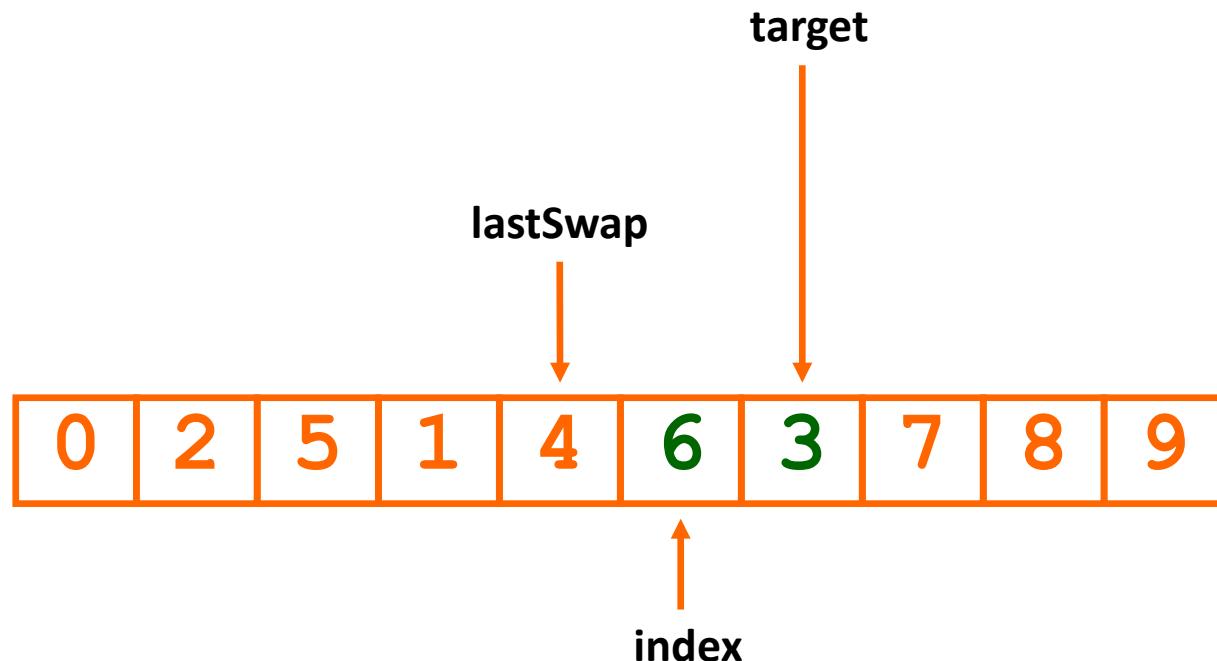
Bubble Sort Animation



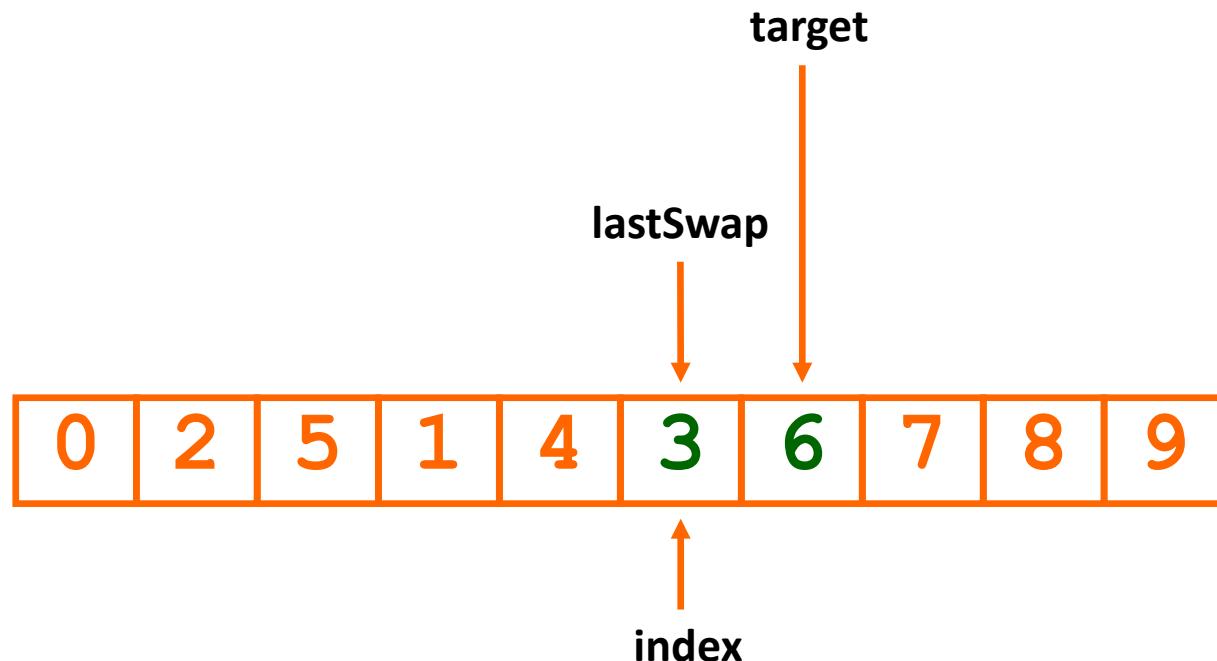
Bubble Sort Animation



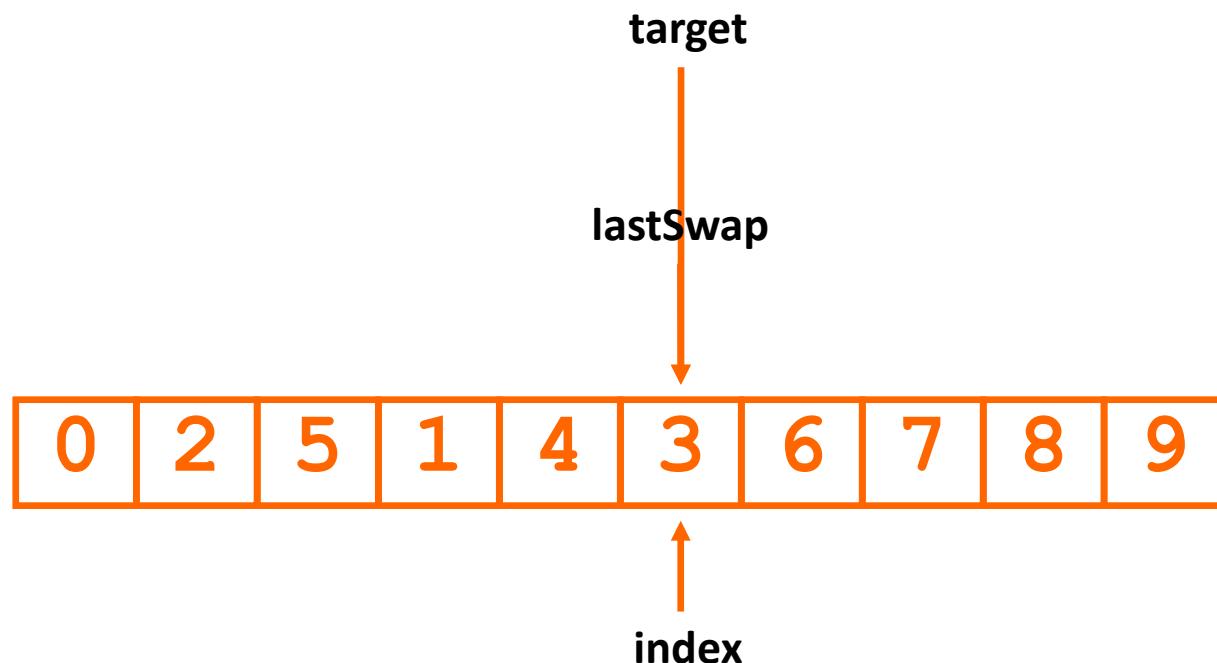
Bubble Sort Animation



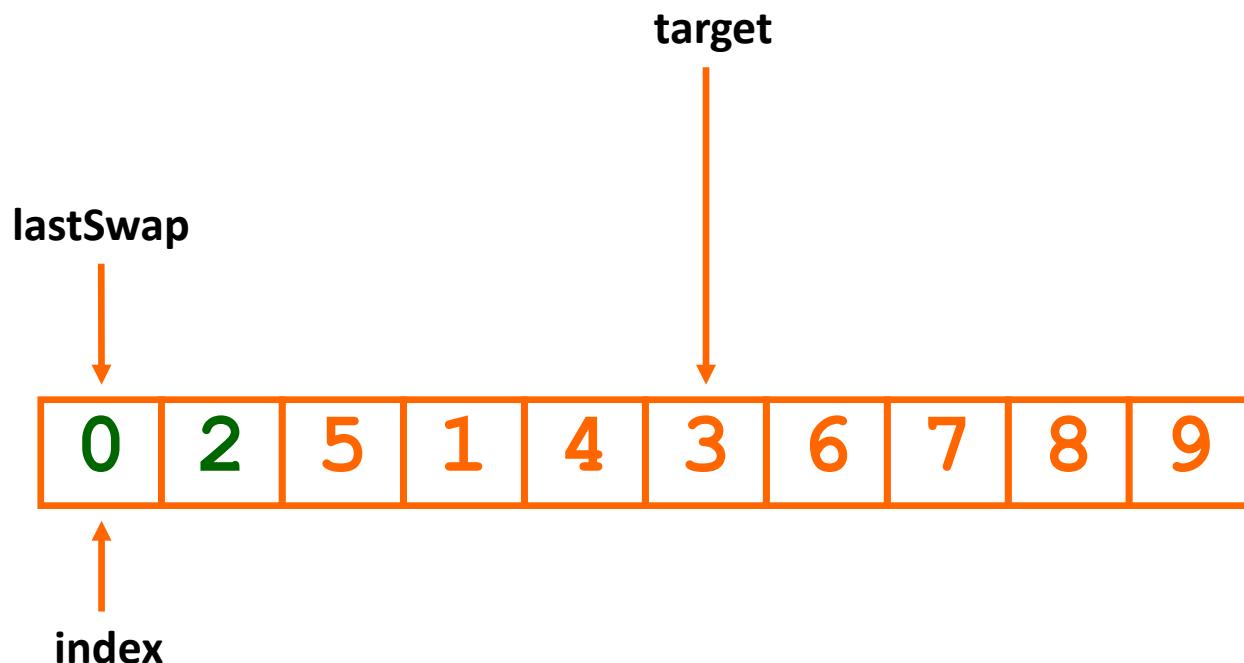
Bubble Sort Animation



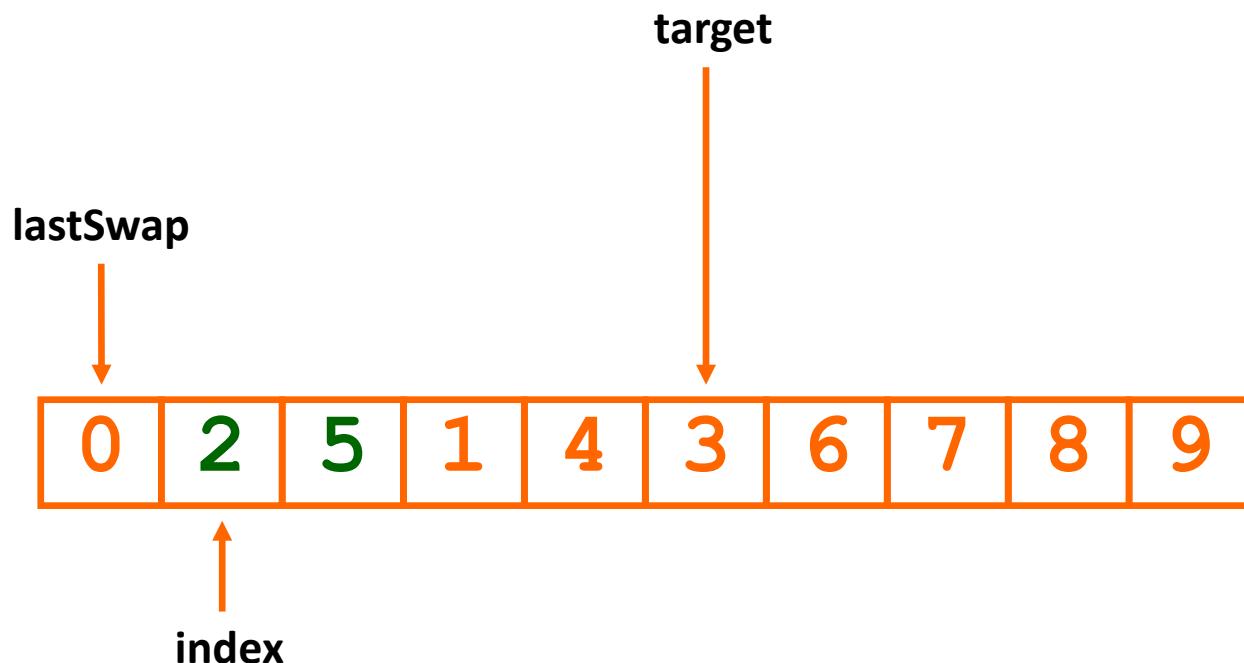
Bubble Sort Animation



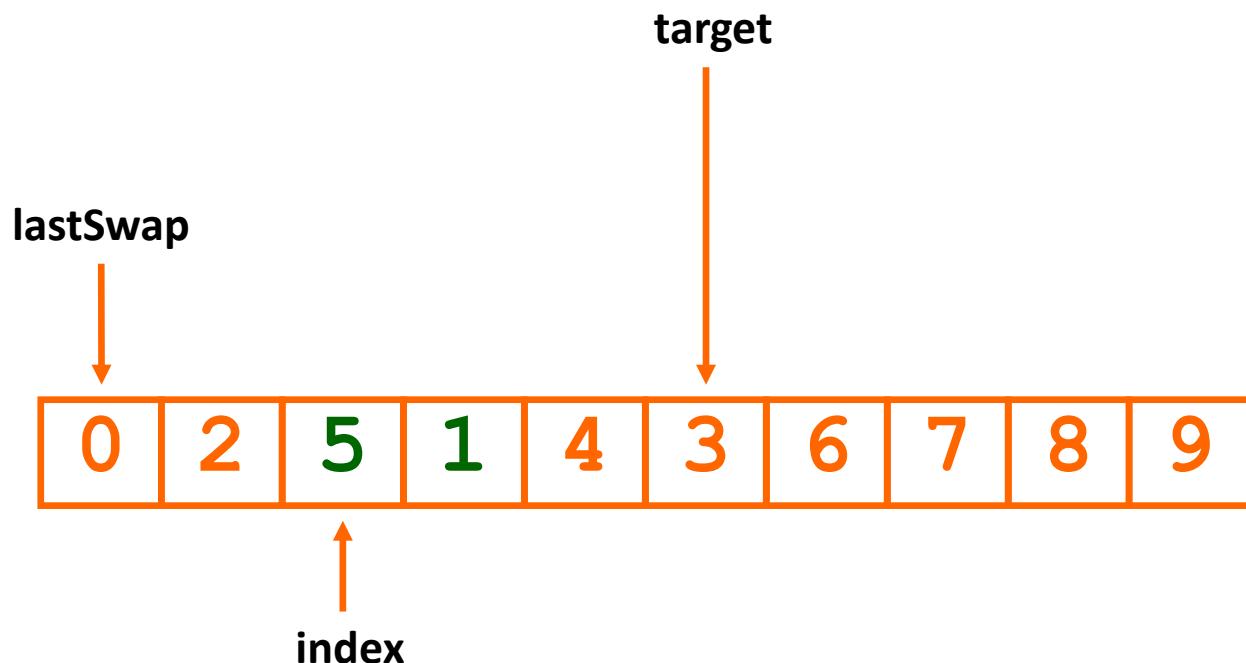
Bubble Sort Animation



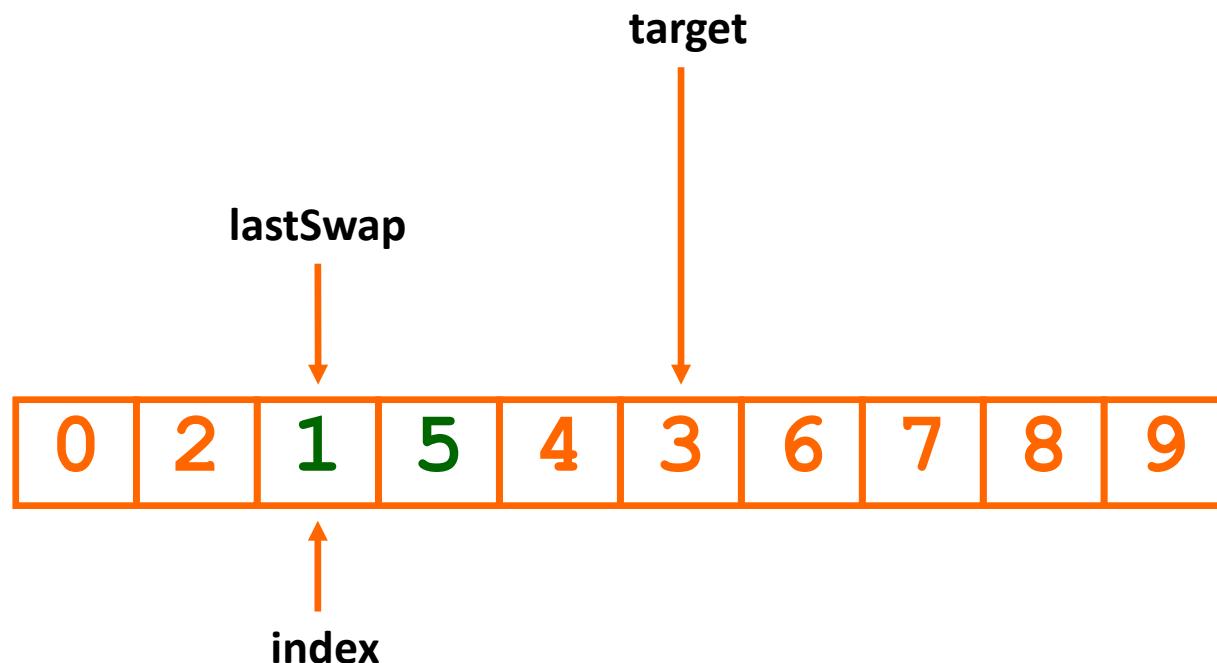
Bubble Sort Animation



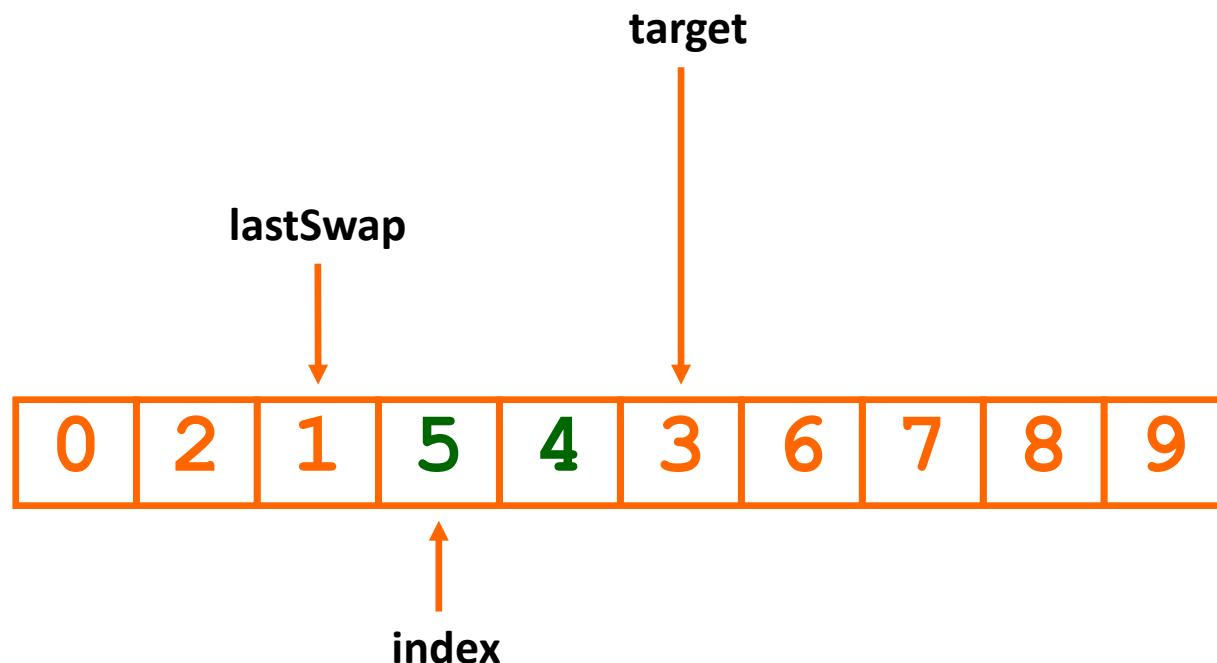
Bubble Sort Animation



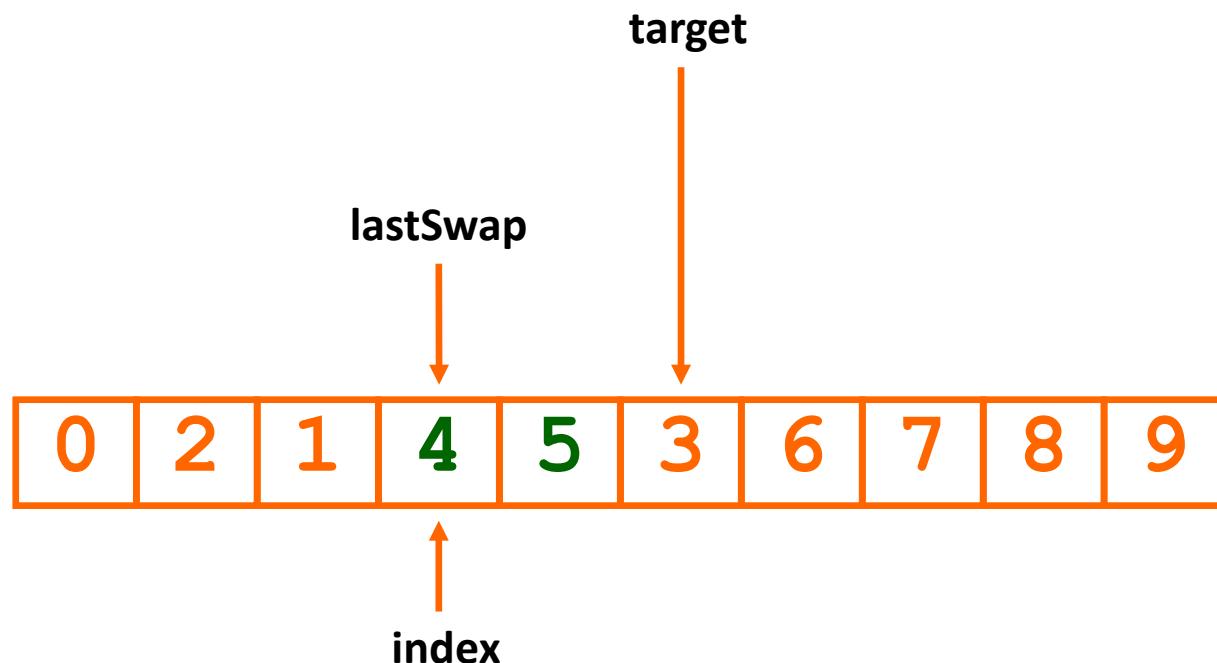
Bubble Sort Animation



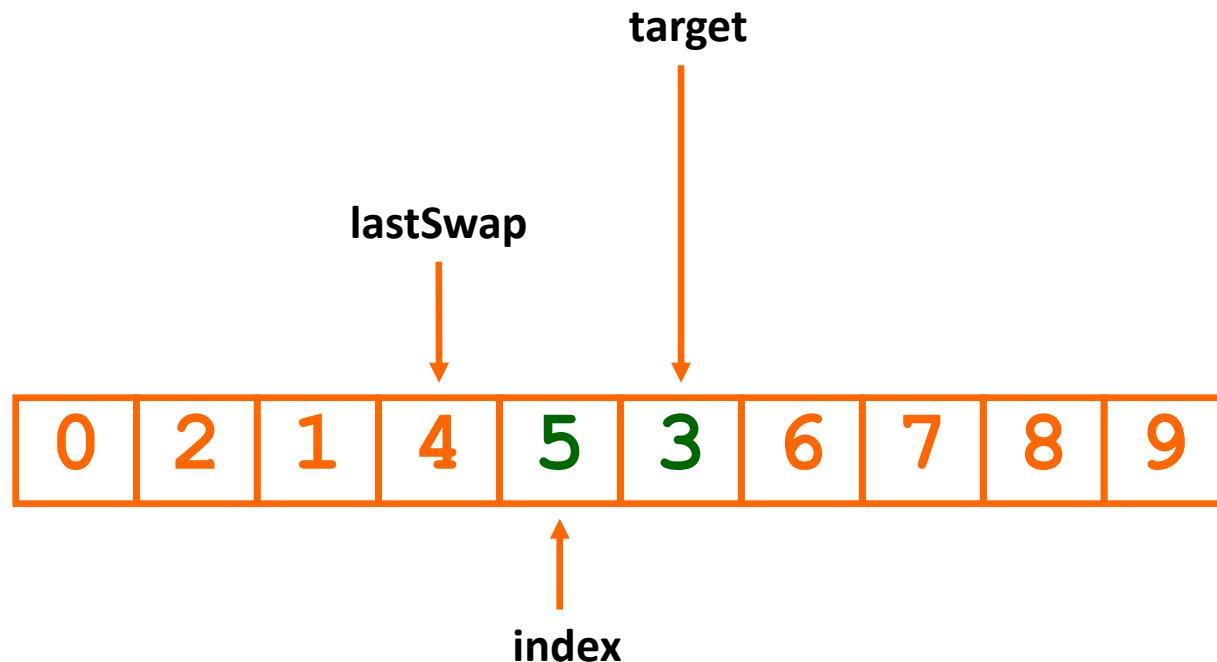
Bubble Sort Animation



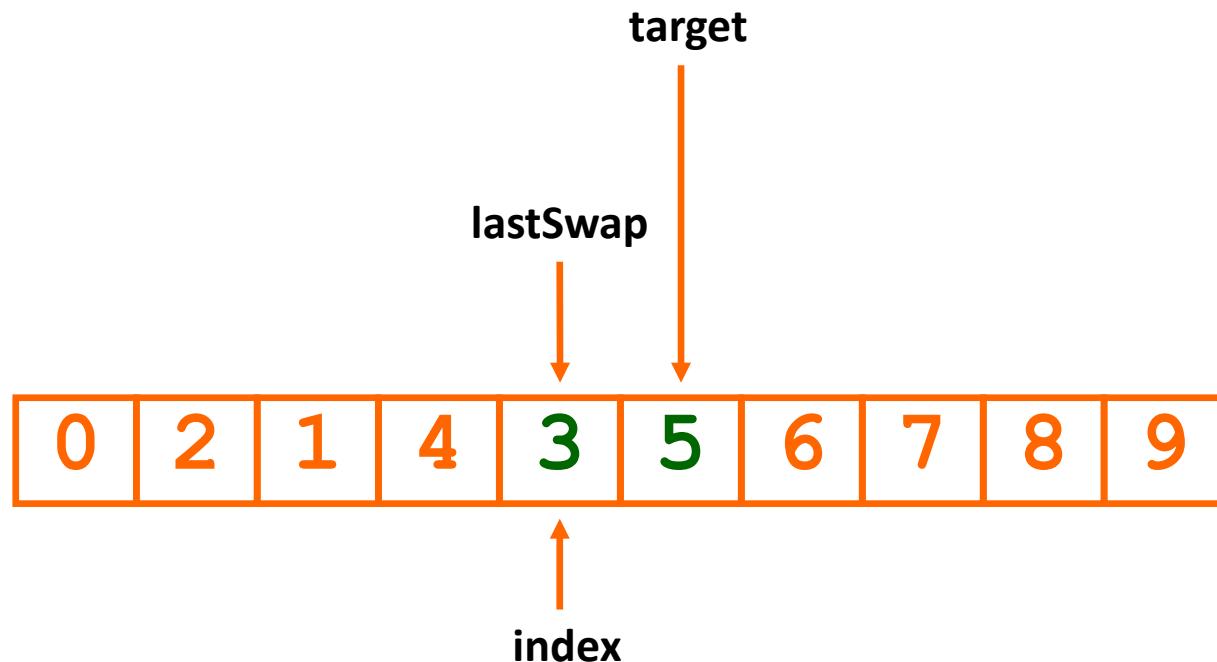
Bubble Sort Animation



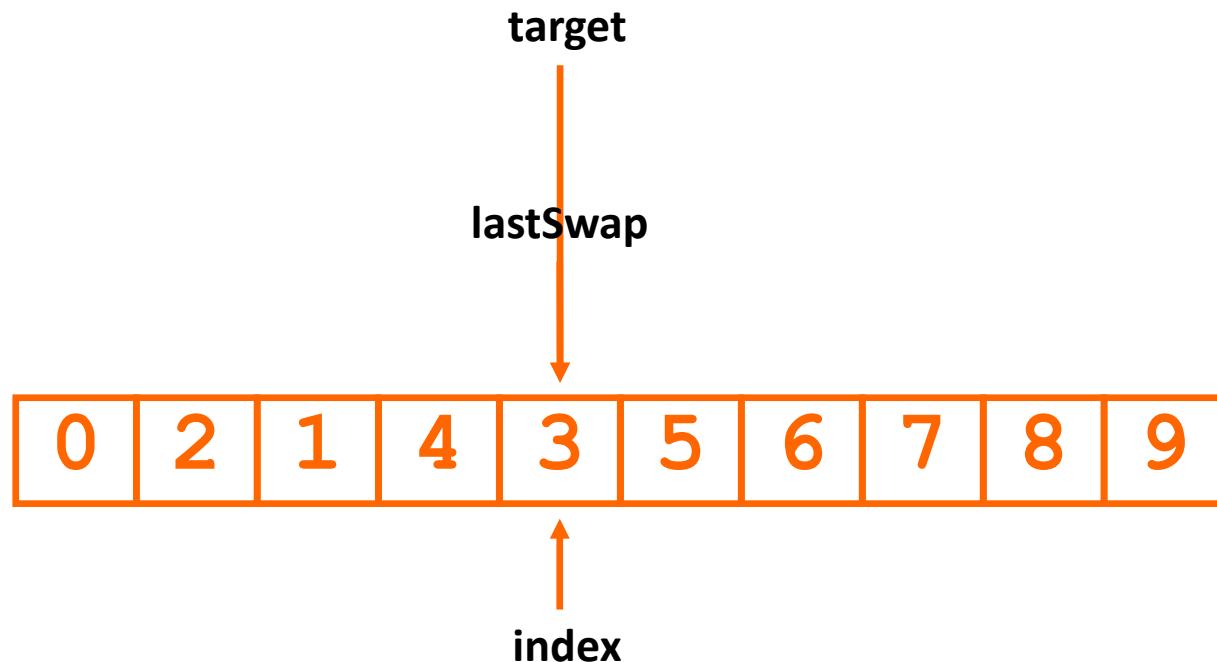
Bubble Sort Animation



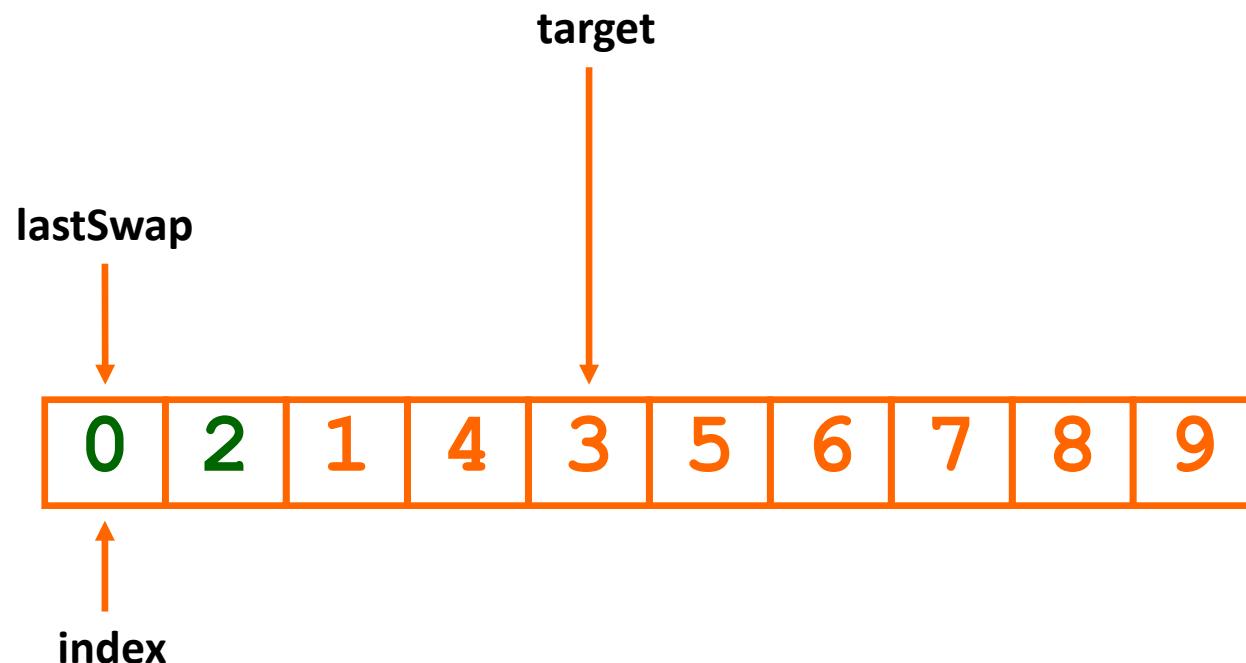
Bubble Sort Animation



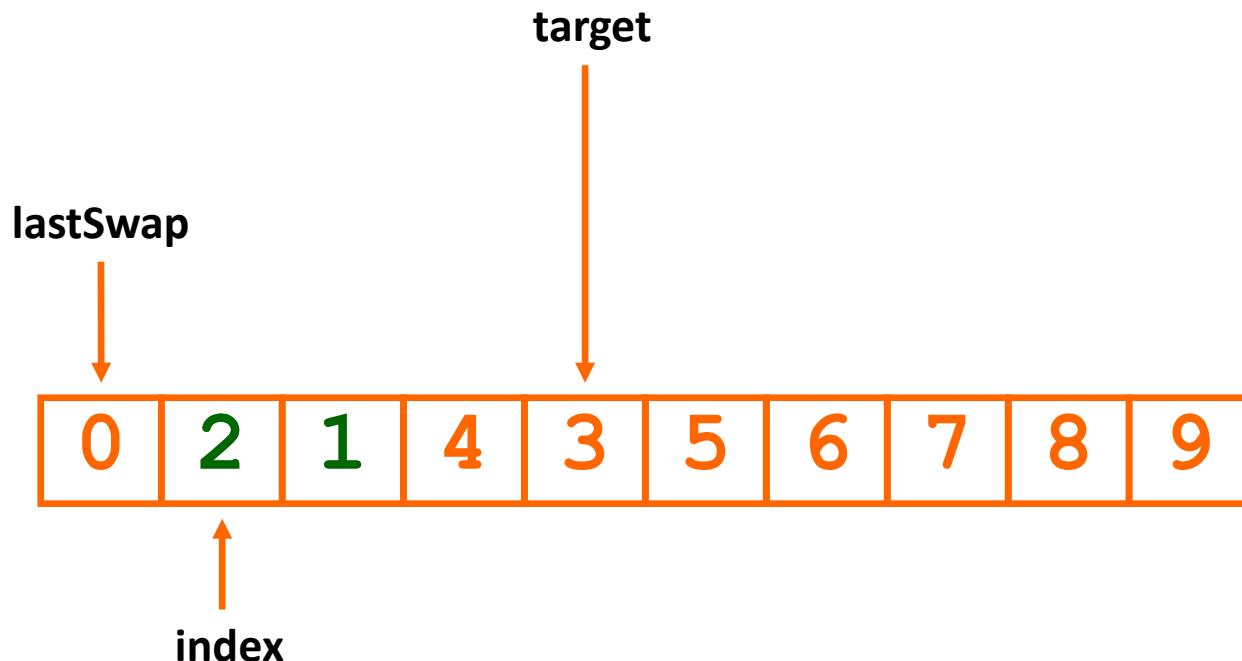
Bubble Sort Animation



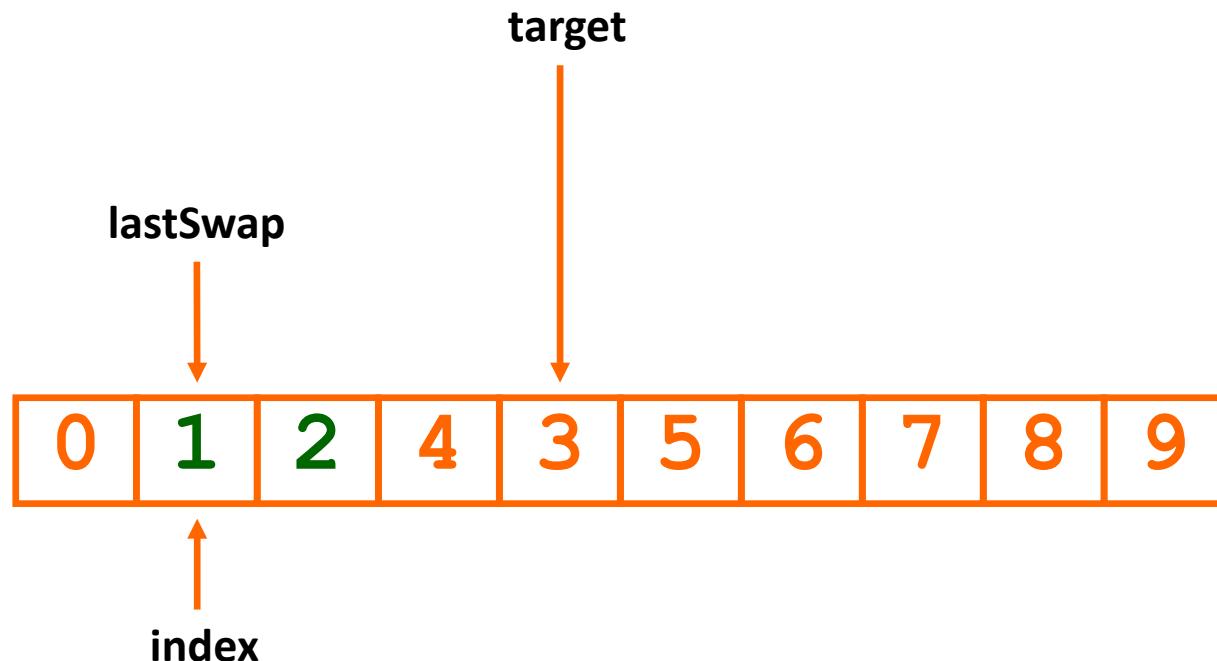
Bubble Sort Animation



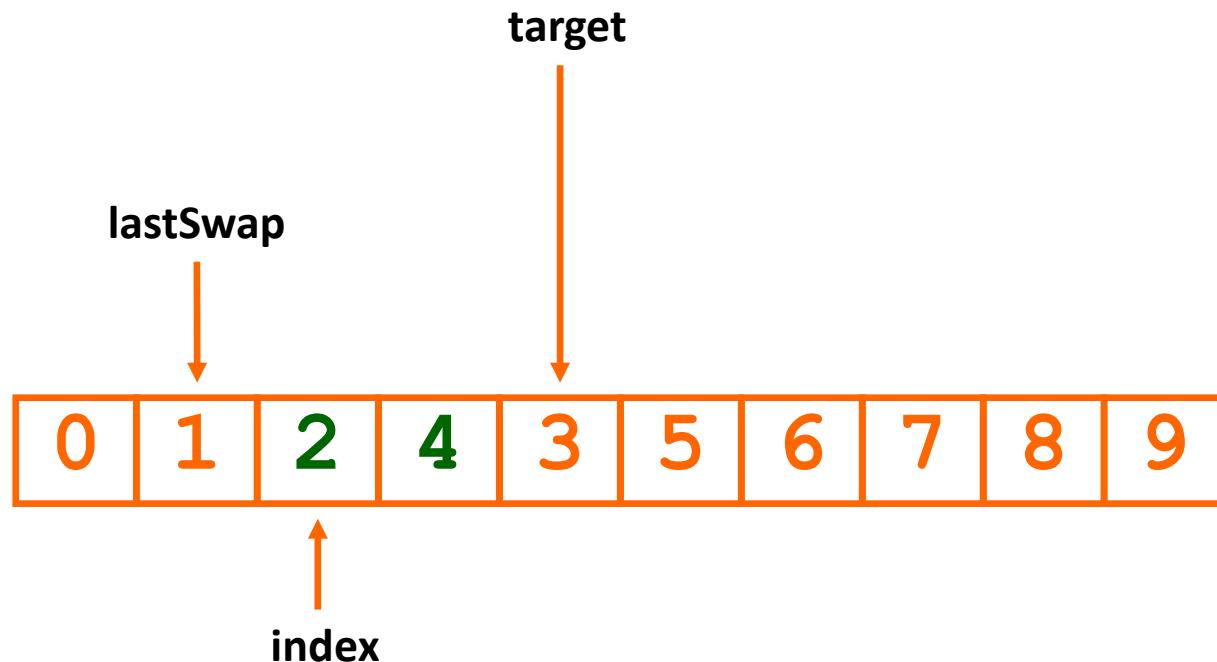
Bubble Sort Animation



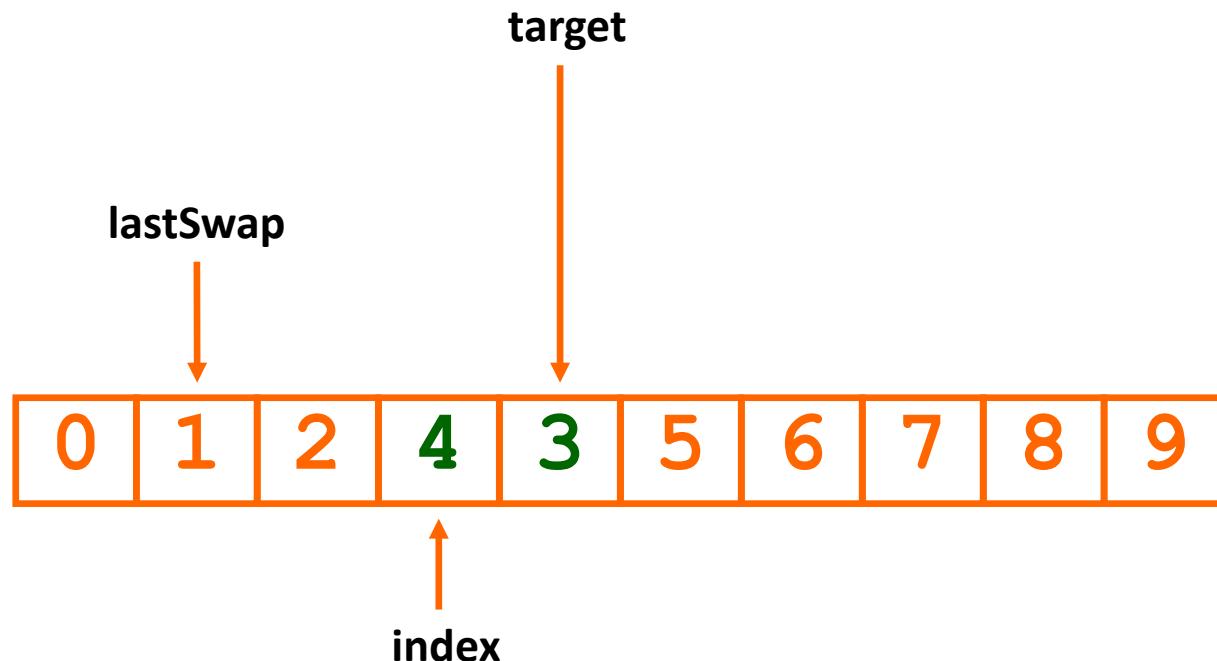
Bubble Sort Animation



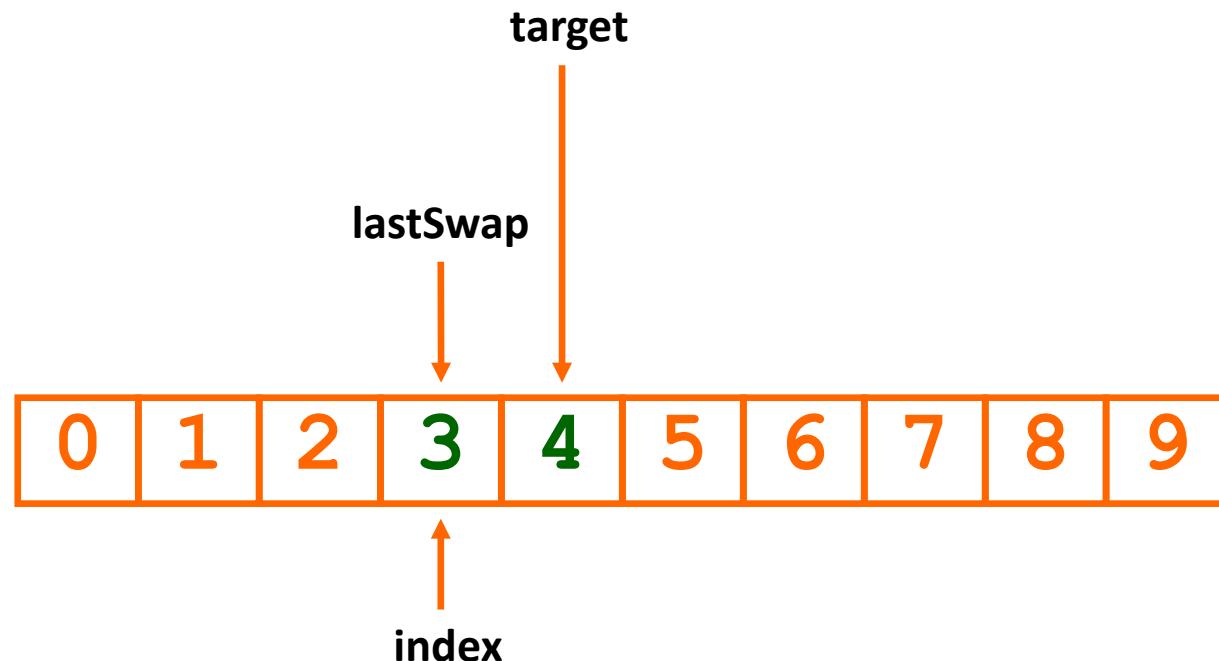
Bubble Sort Animation



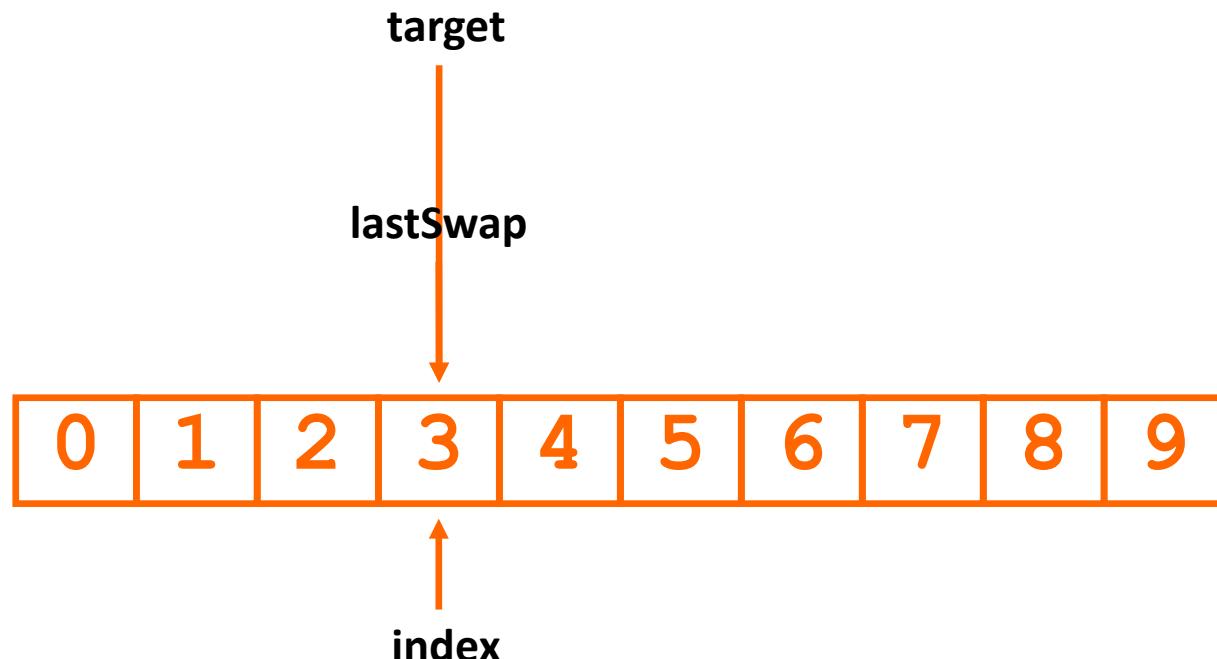
Bubble Sort Animation



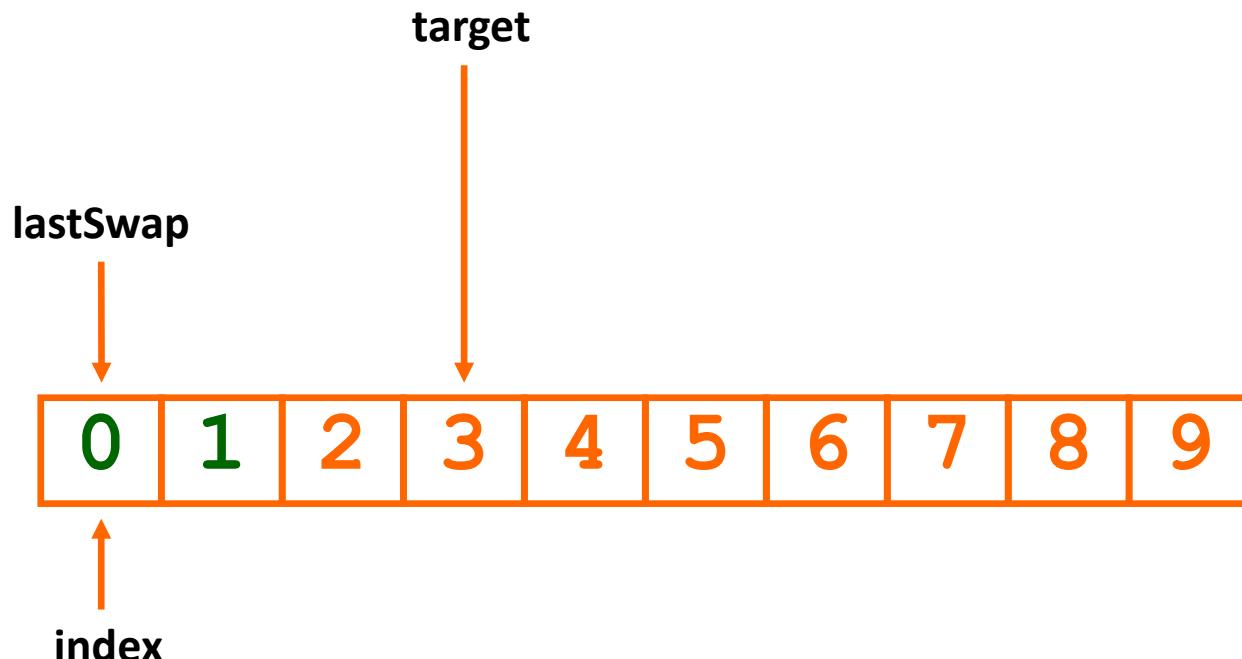
Bubble Sort Animation



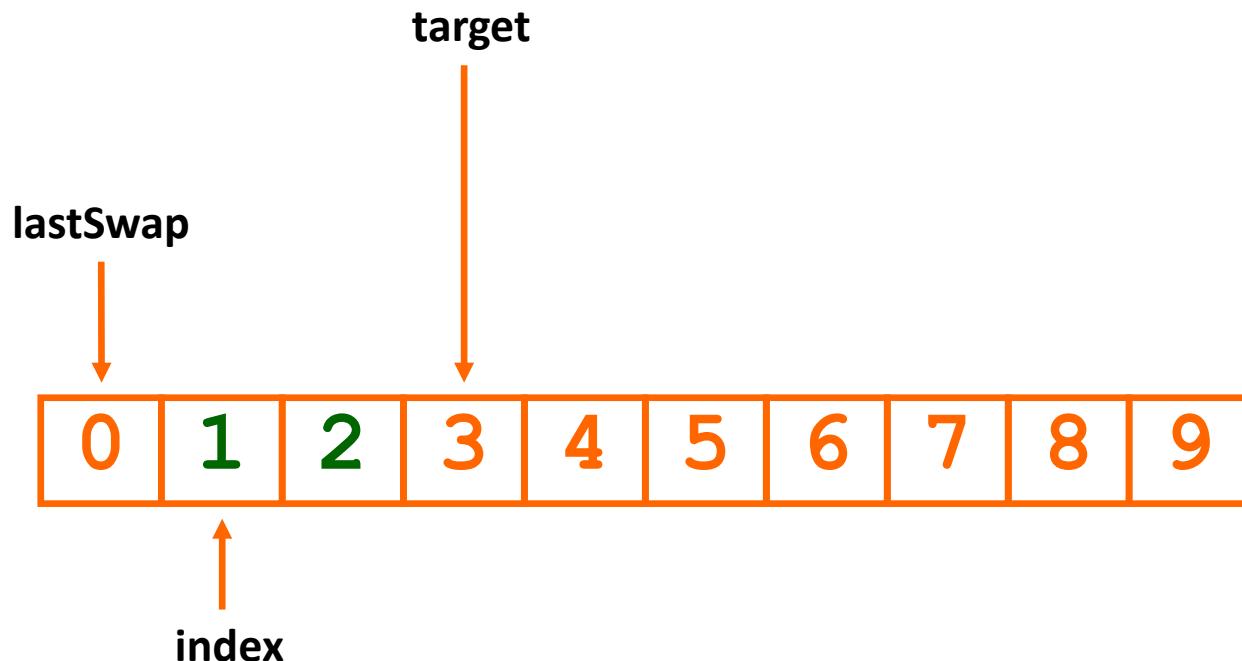
Bubble Sort Animation



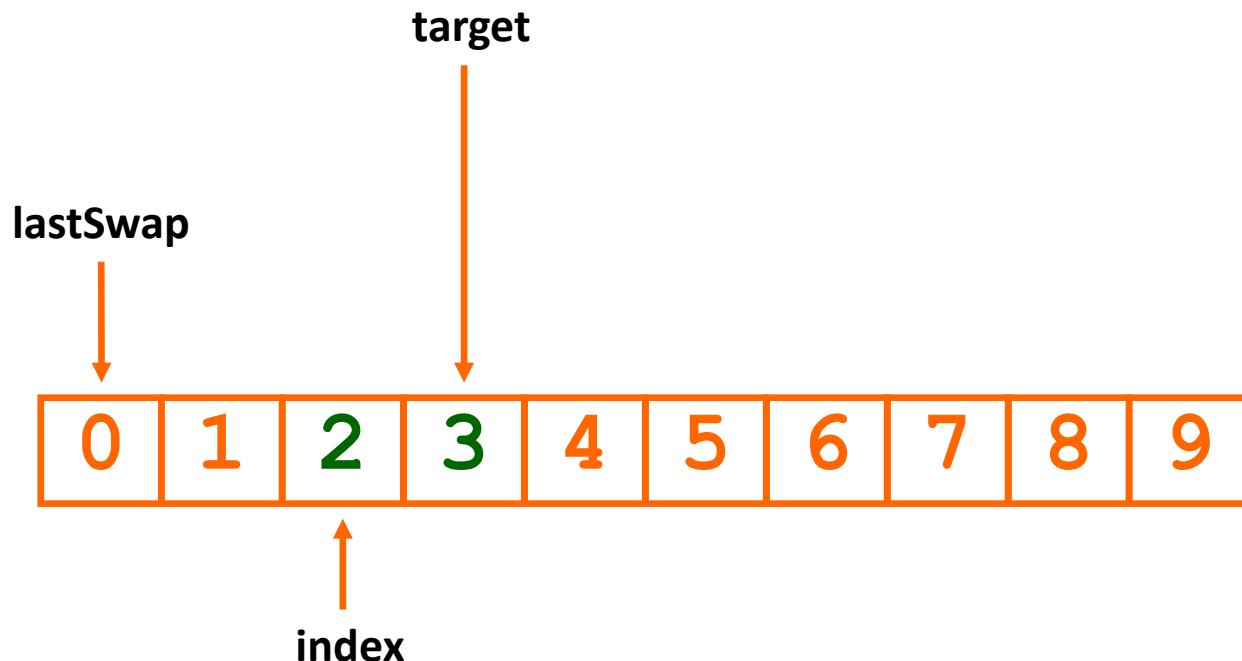
Bubble Sort Animation



Bubble Sort Animation

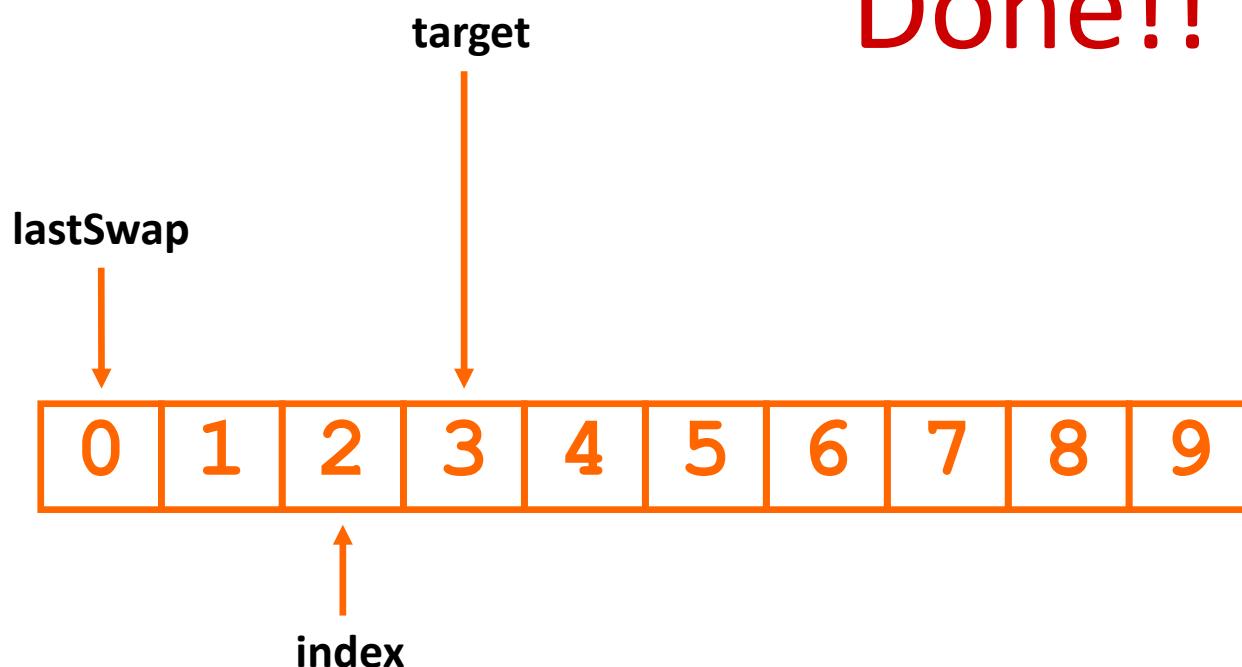


Bubble Sort Animation



Bubble Sort Animation

Done!!



Merge Sort [1]

- Merge sort uses the divide and conquer algorithmic strategy.
- It has complexity $O(n \log n)$ 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).

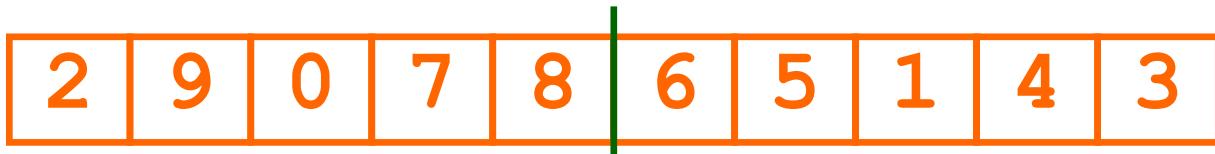
Merge Sort Algorithm

- 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

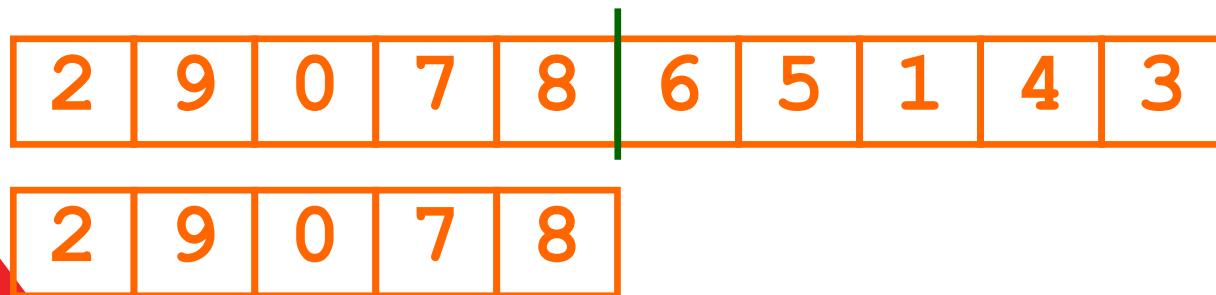
Merge Sort Animation



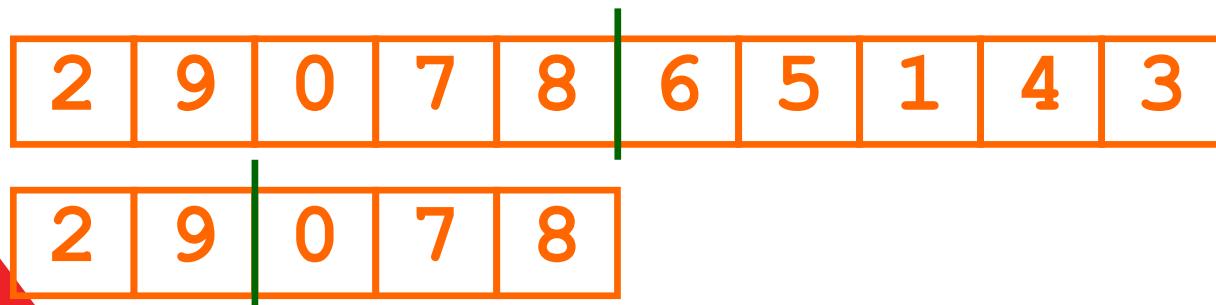
Merge Sort Animation



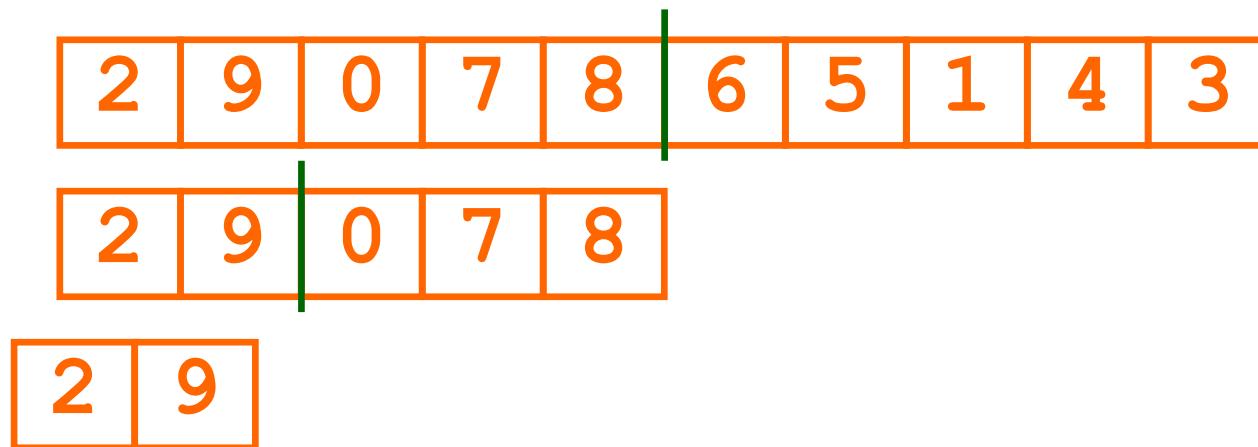
Merge Sort Animation



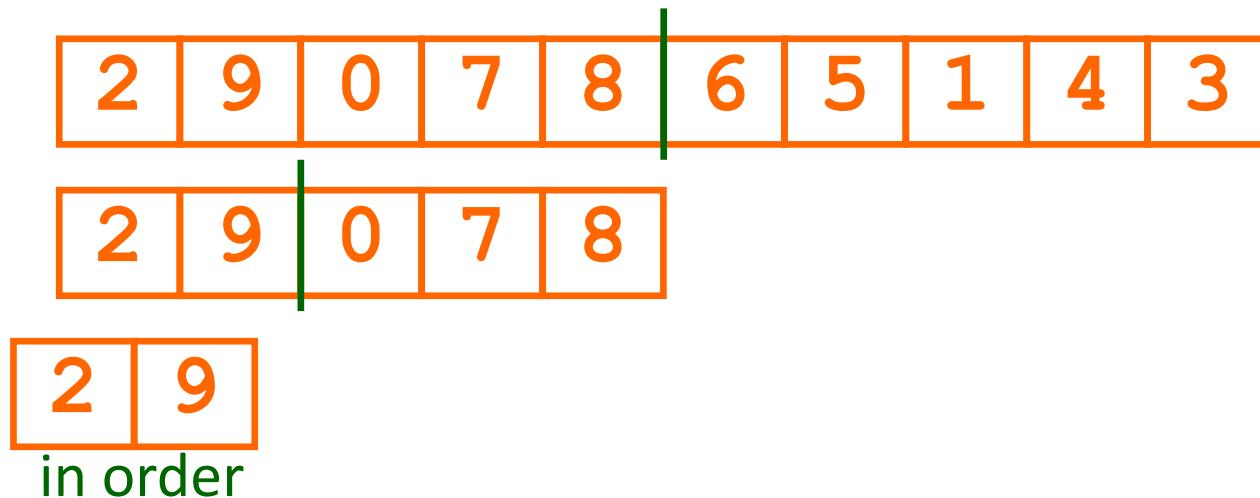
Merge Sort Animation



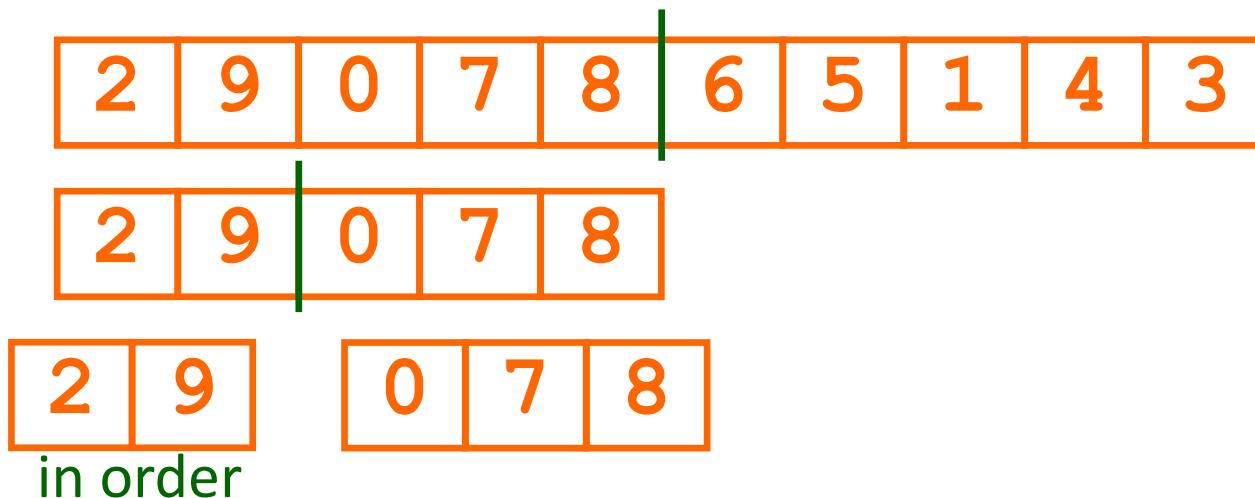
Merge Sort Animation



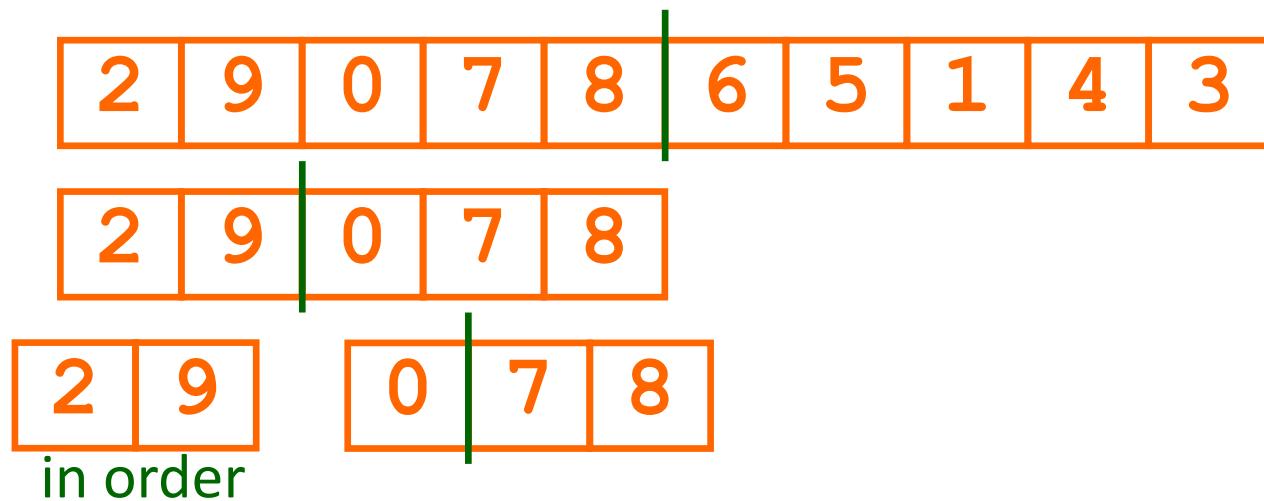
Merge Sort Animation



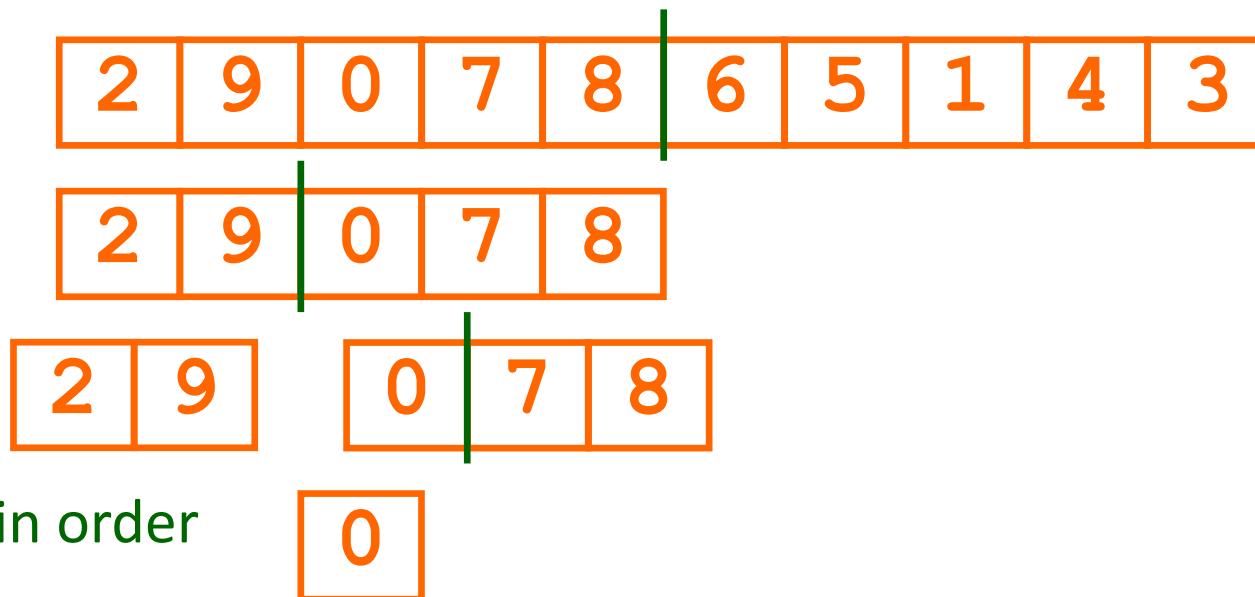
Merge Sort Animation



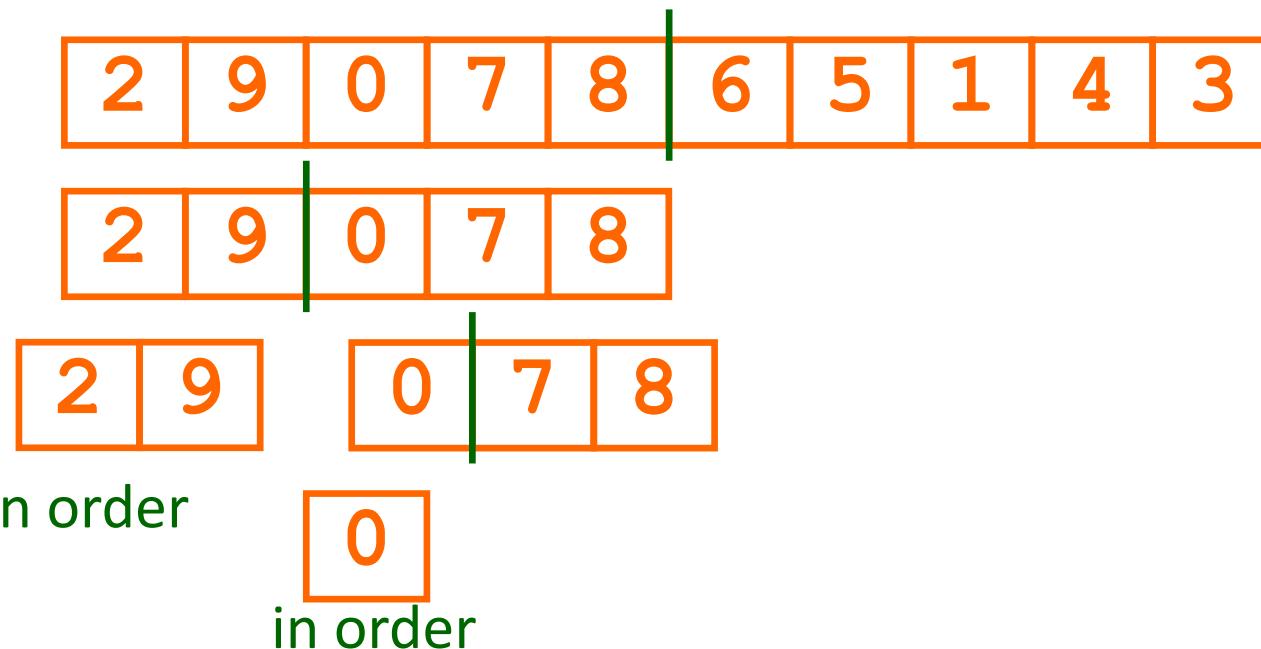
Merge Sort Animation



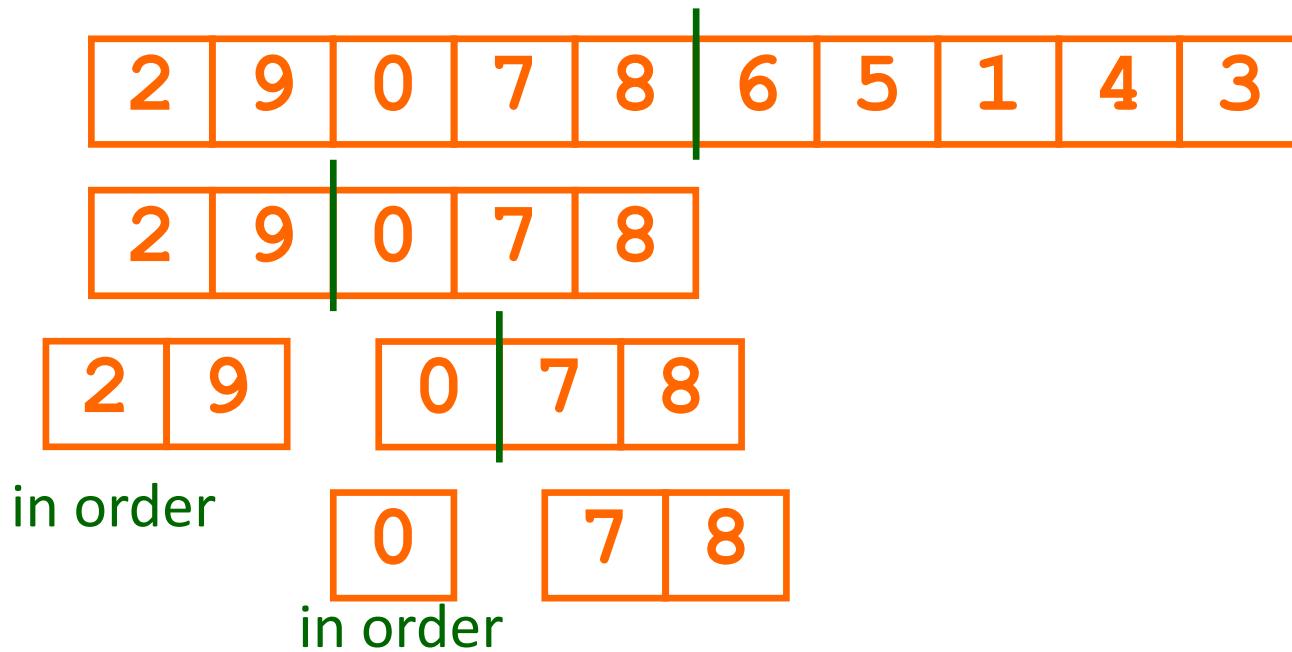
Merge Sort Animation



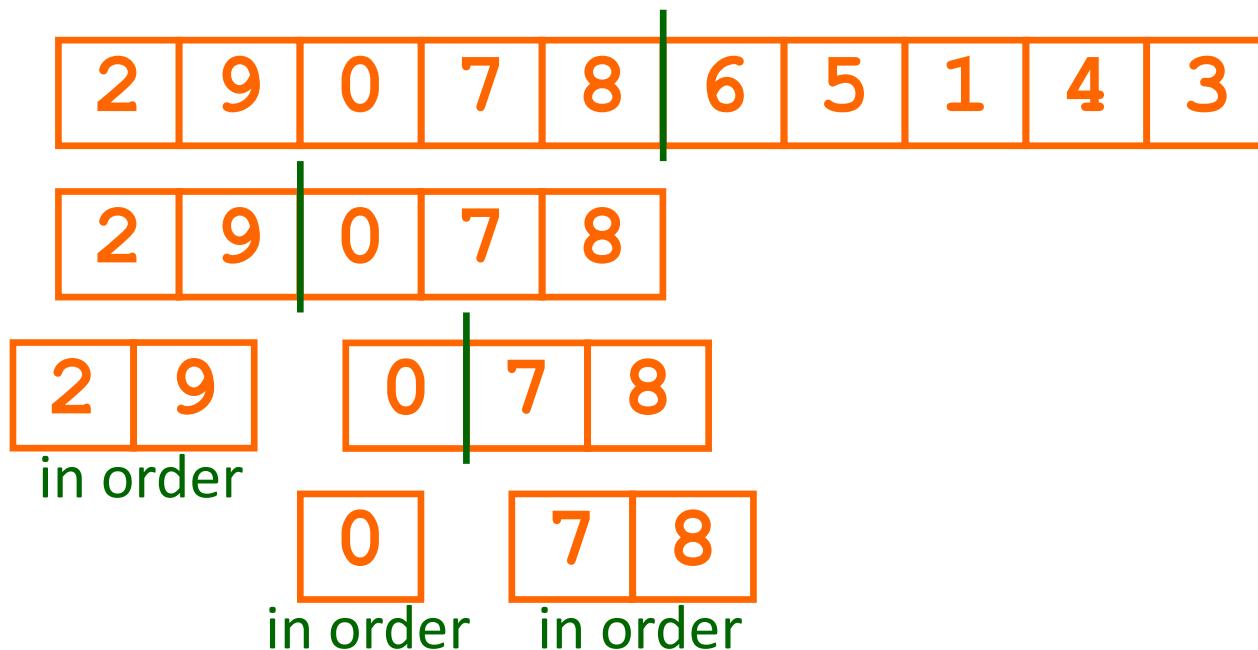
Merge Sort Animation



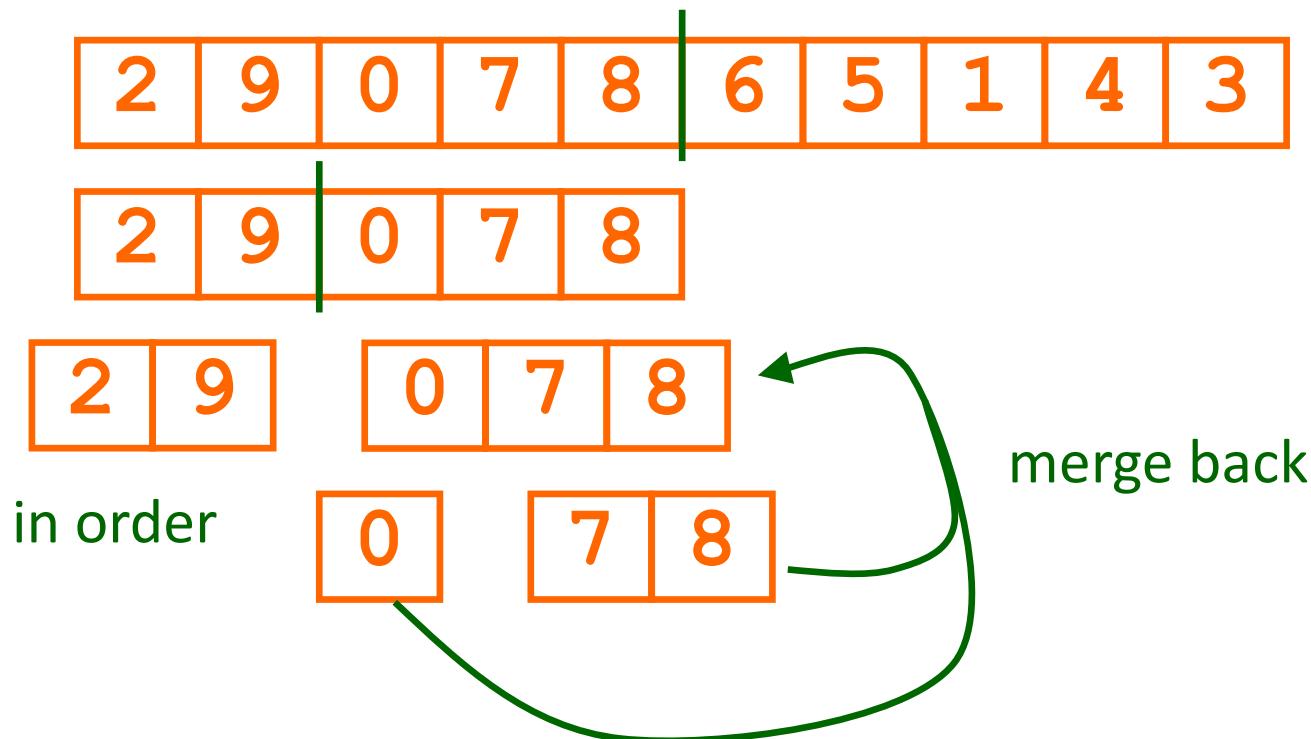
Merge Sort Animation



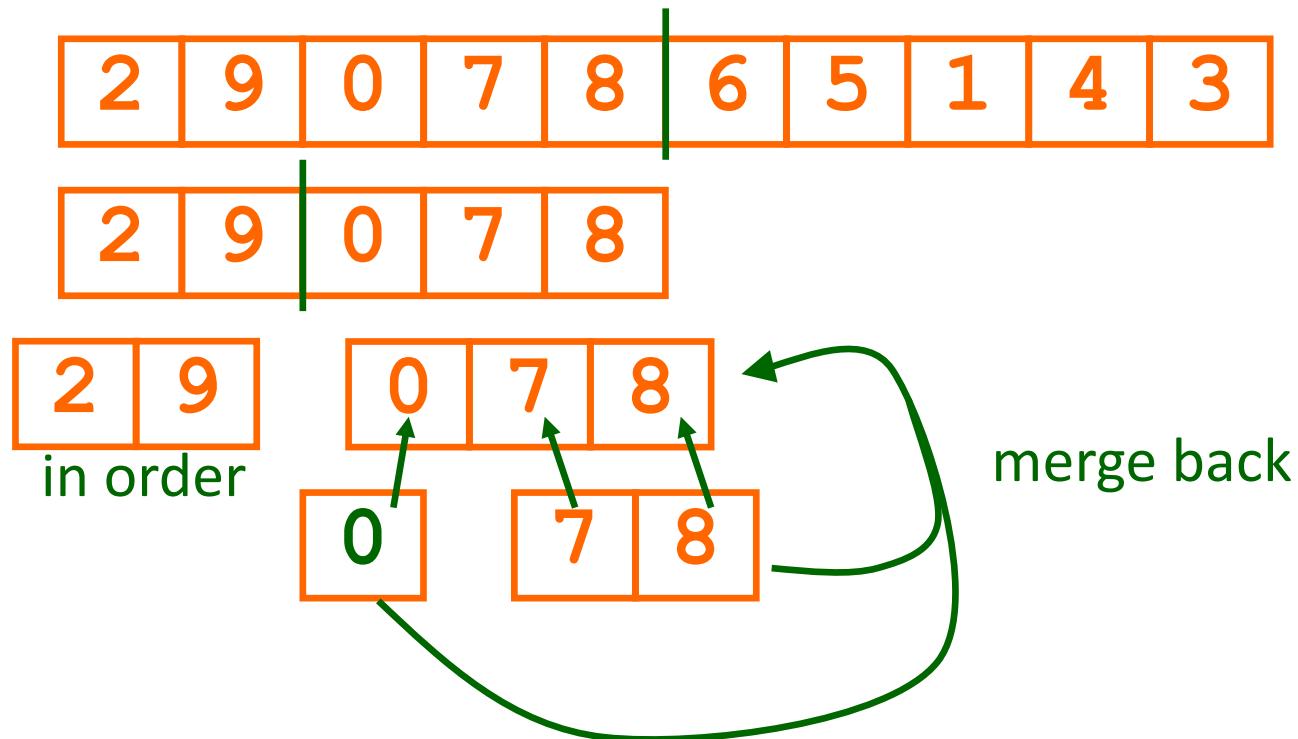
Merge Sort Animation



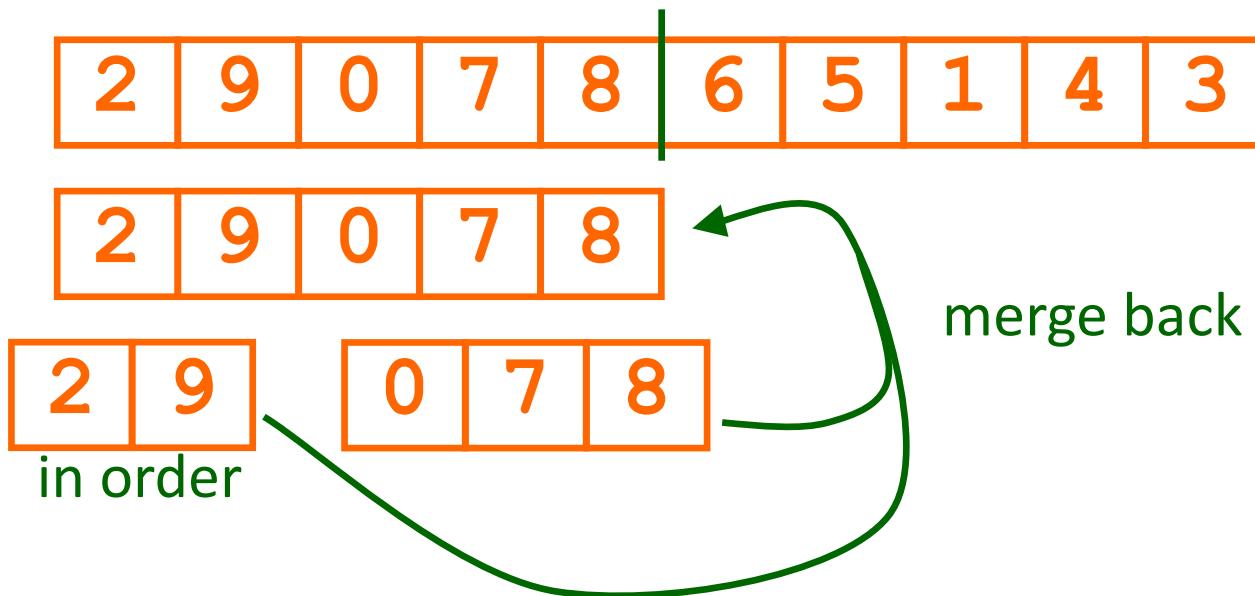
Merge Sort Animation



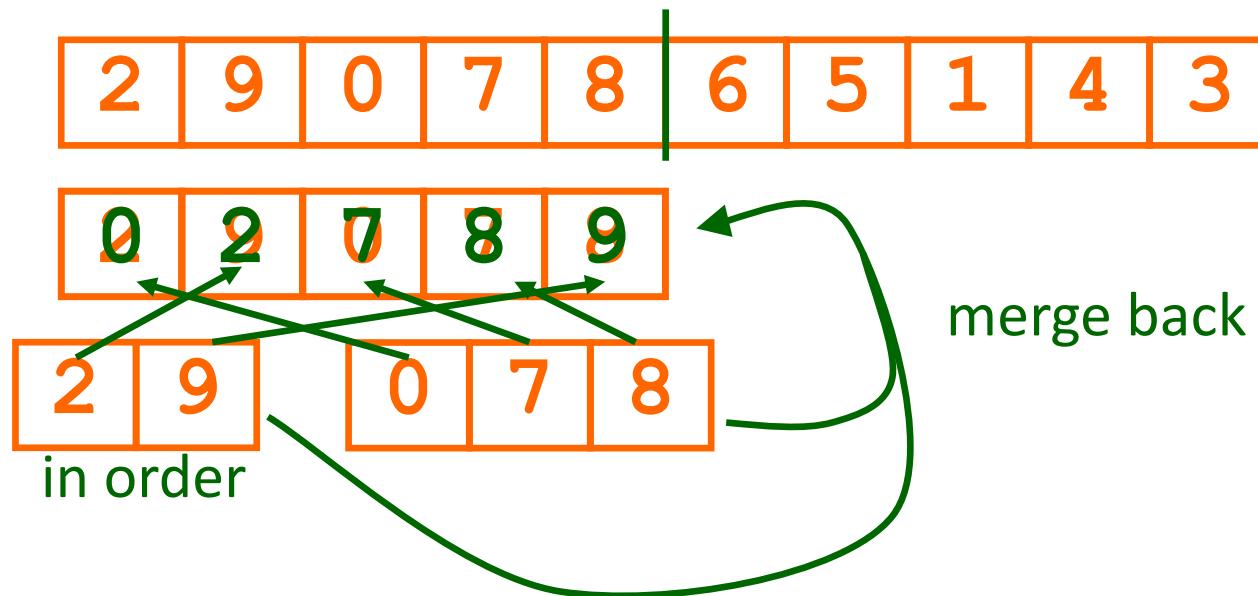
Merge Sort Animation



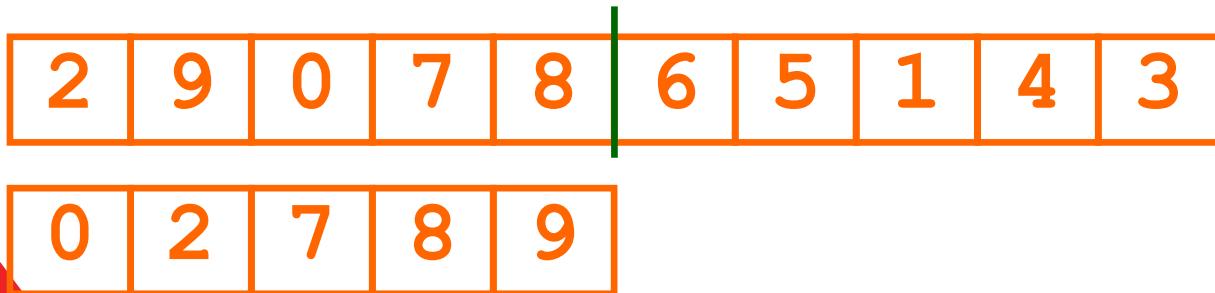
Merge Sort Animation



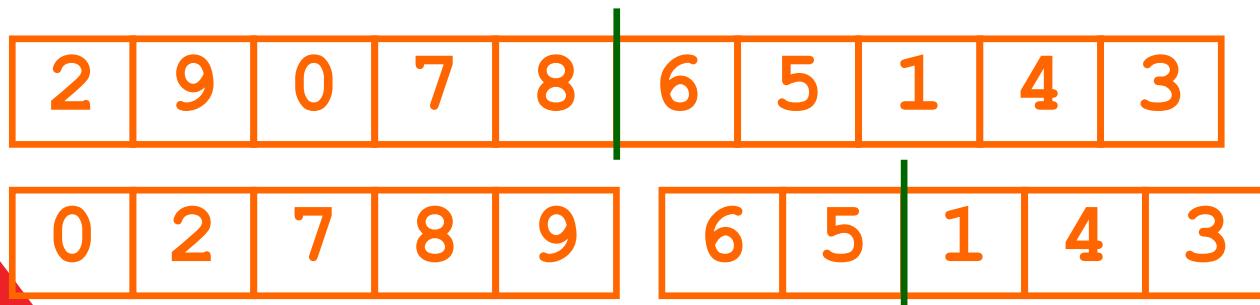
Merge Sort Animation



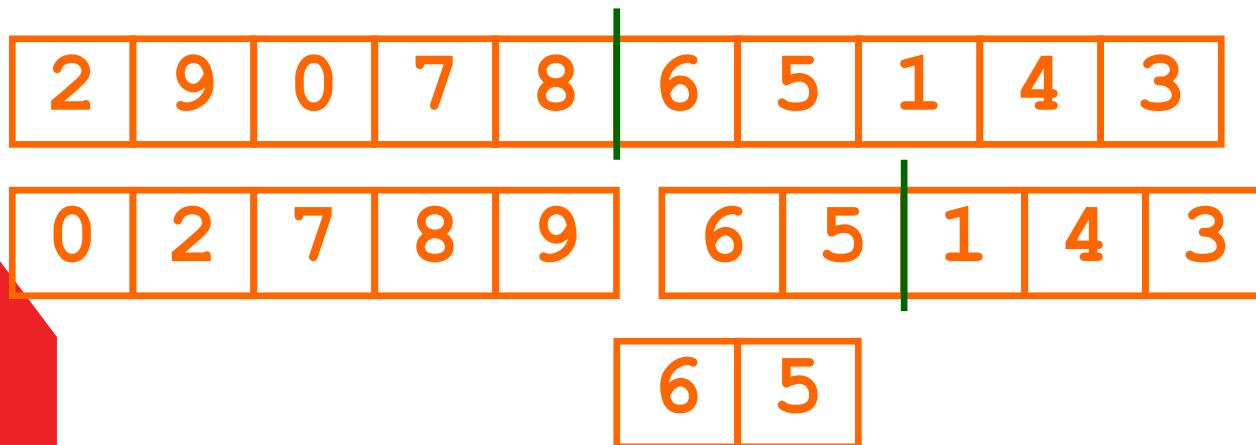
Merge Sort Animation



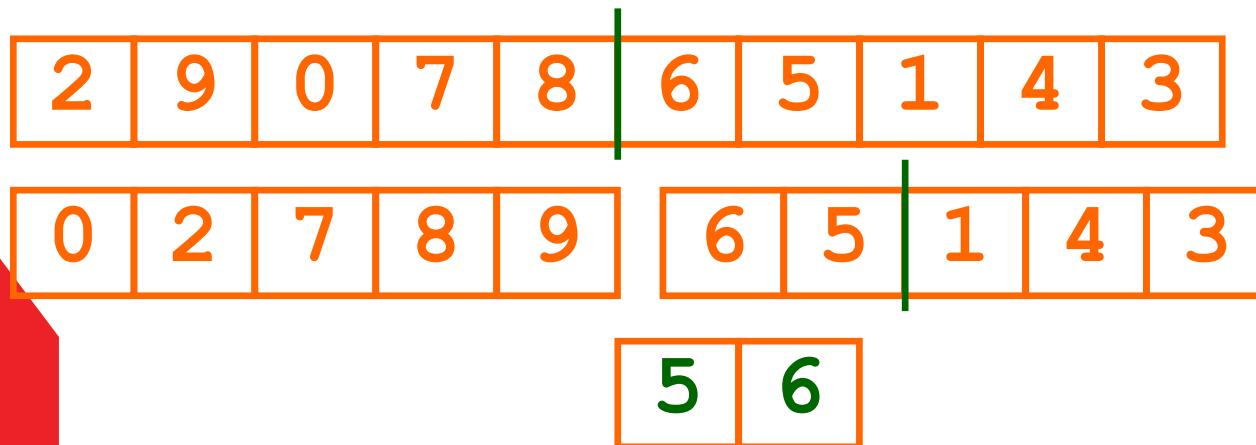
Merge Sort Animation



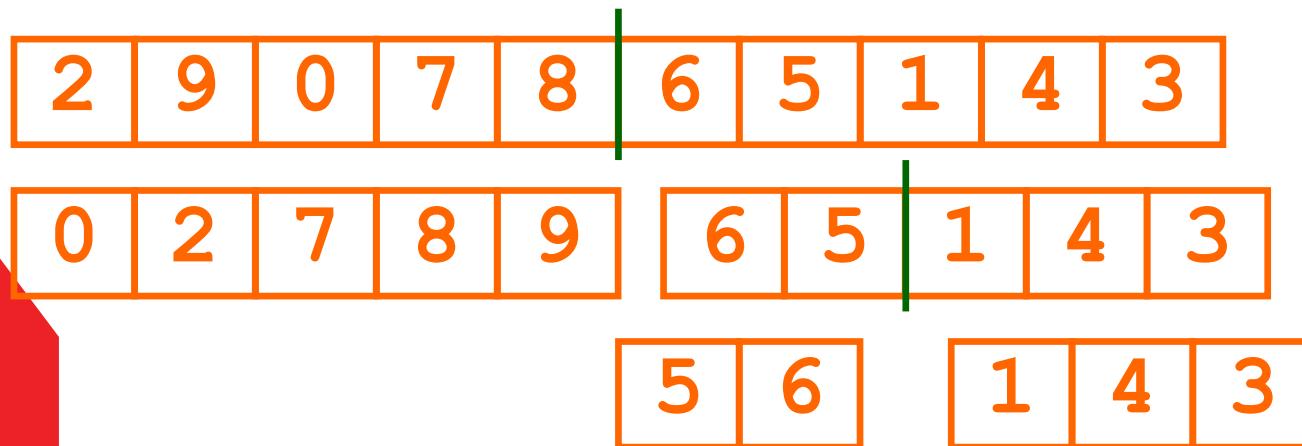
Merge Sort Animation



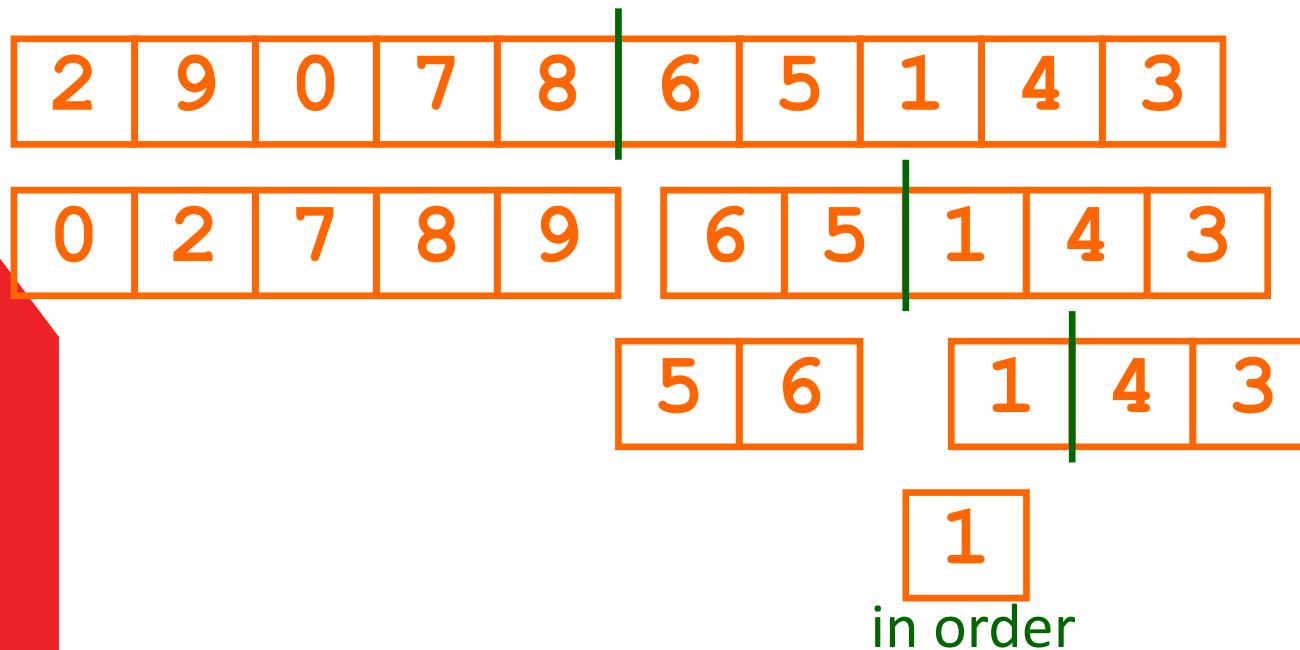
Merge Sort Animation



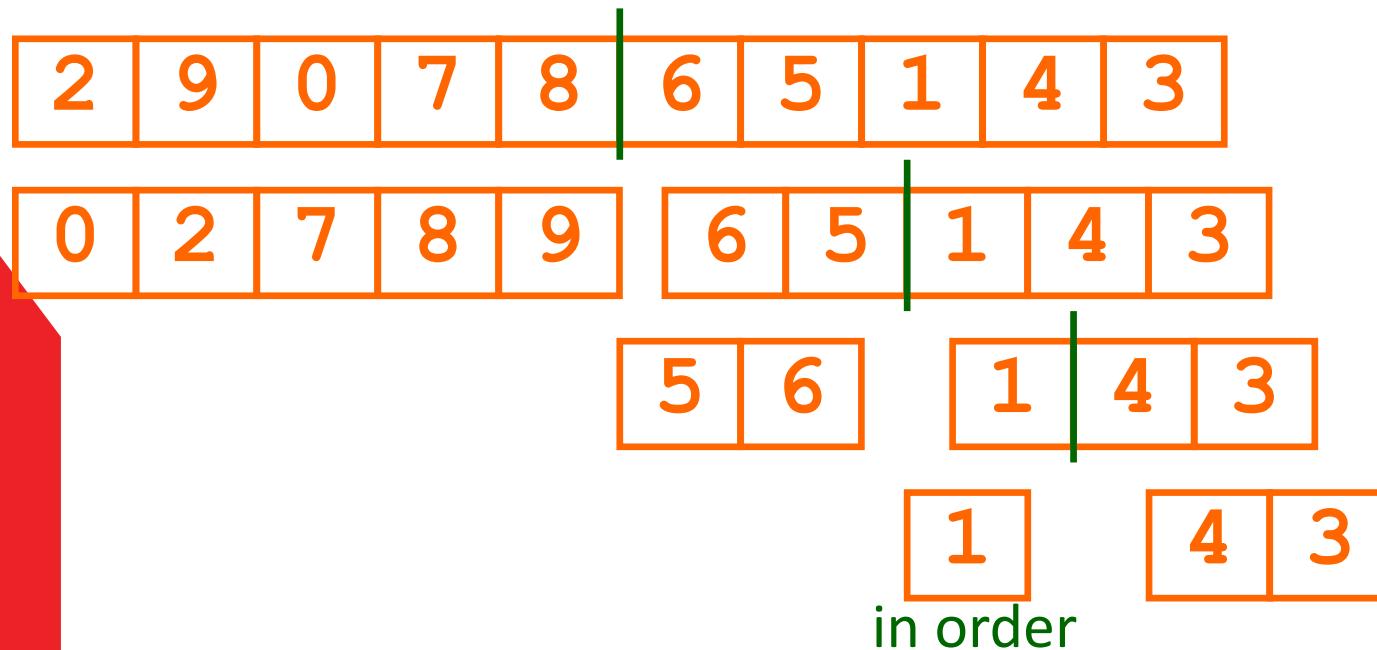
Merge Sort Animation



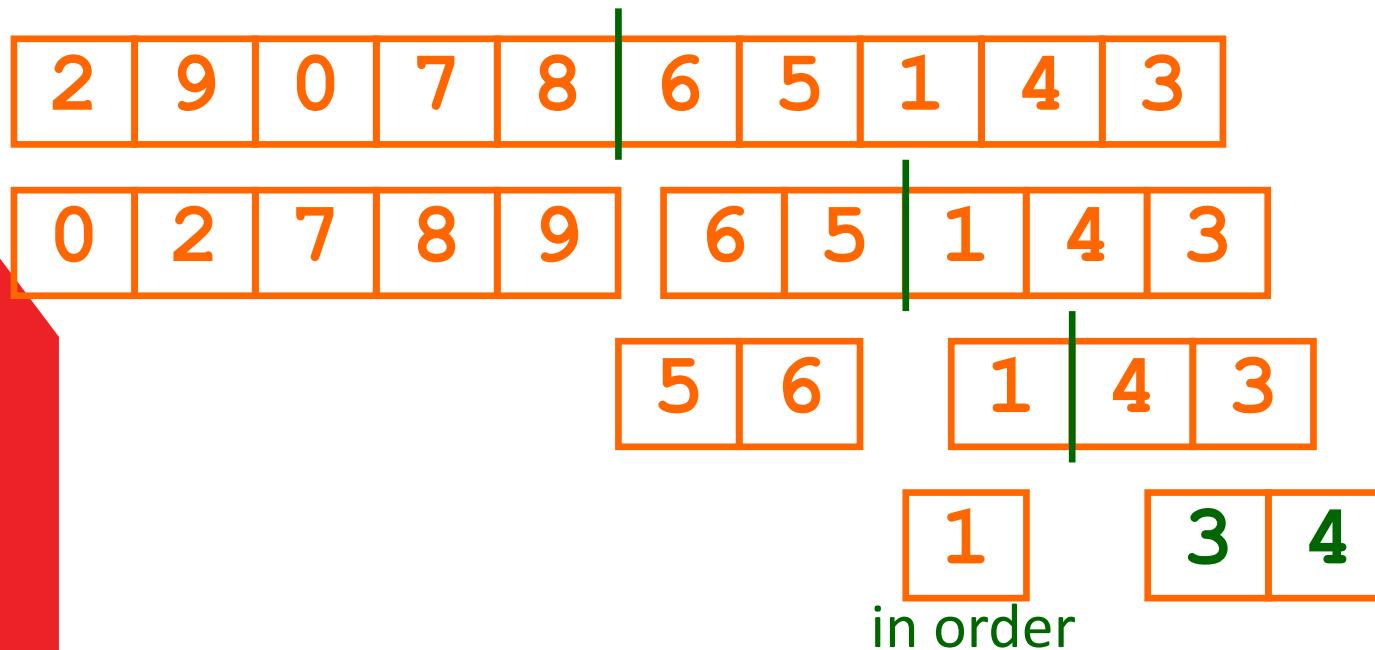
Merge Sort Animation



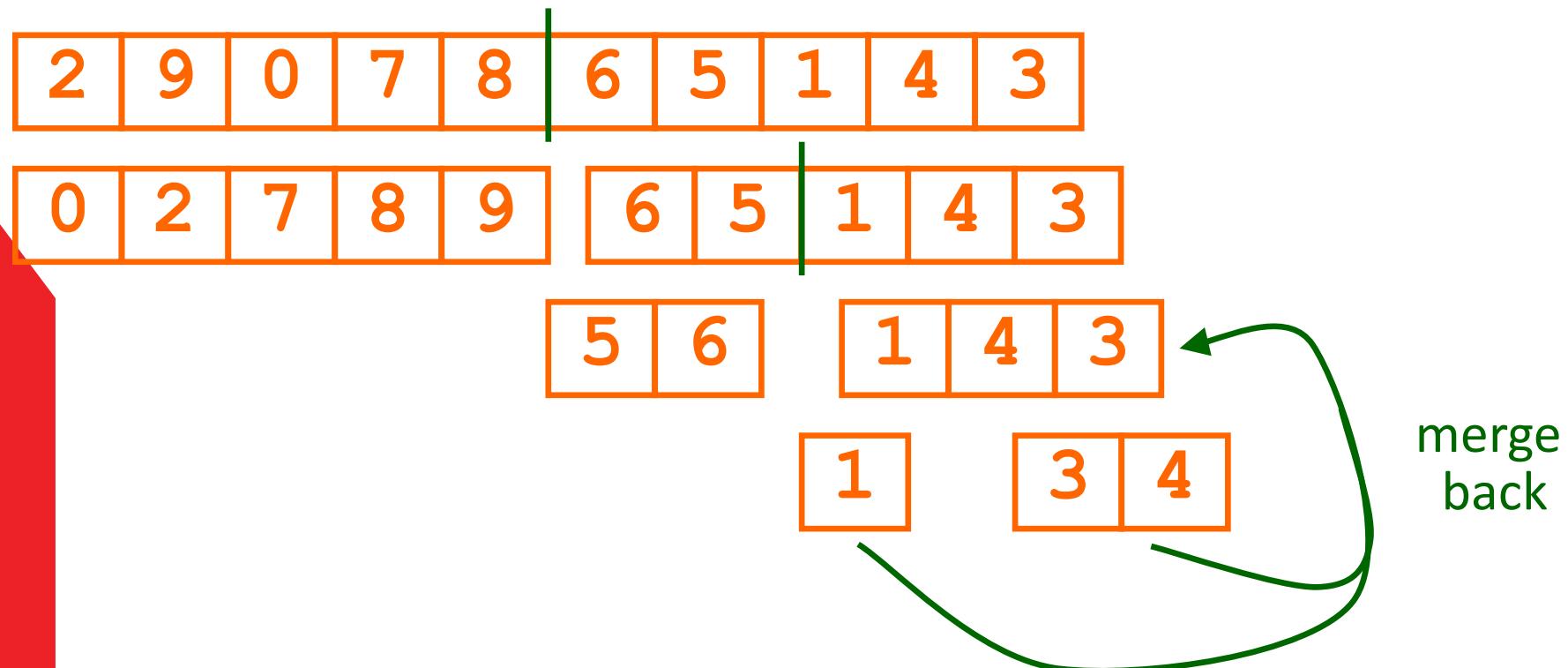
Merge Sort Animation



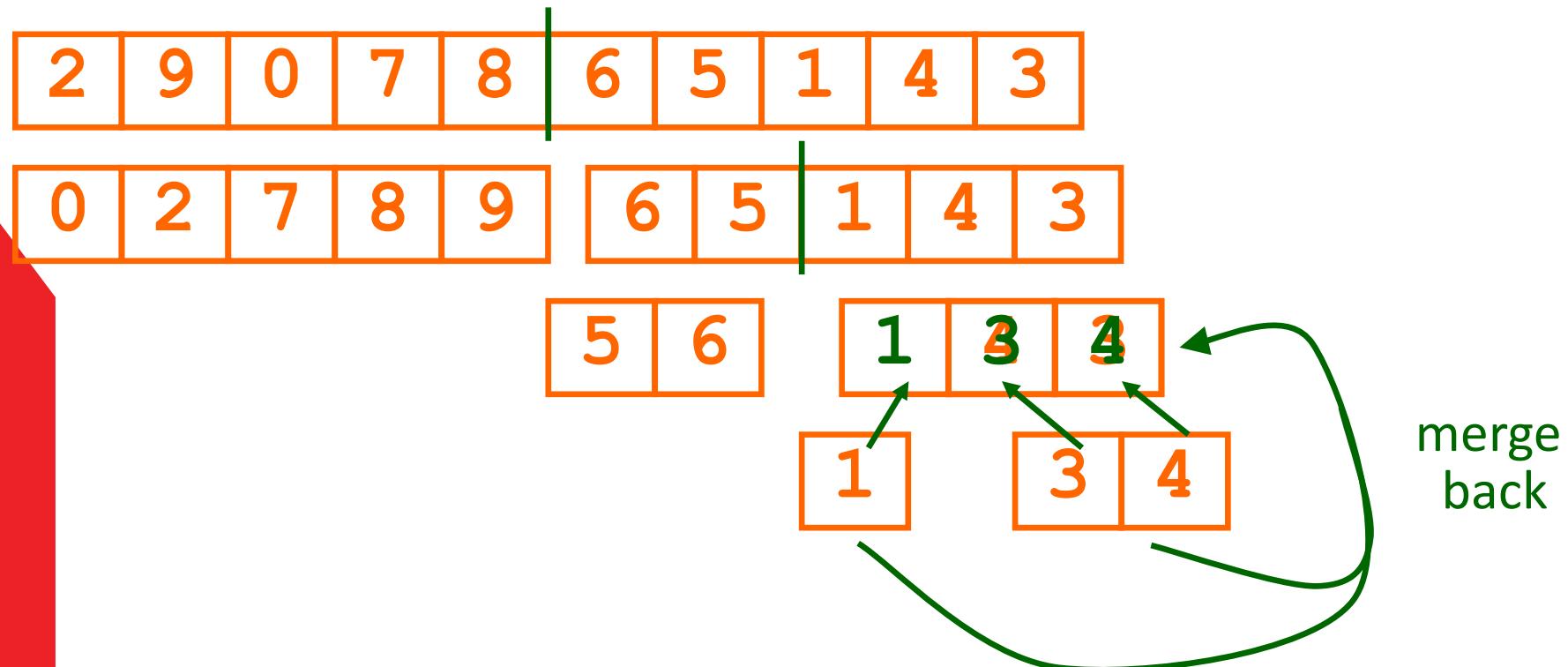
Merge Sort Animation



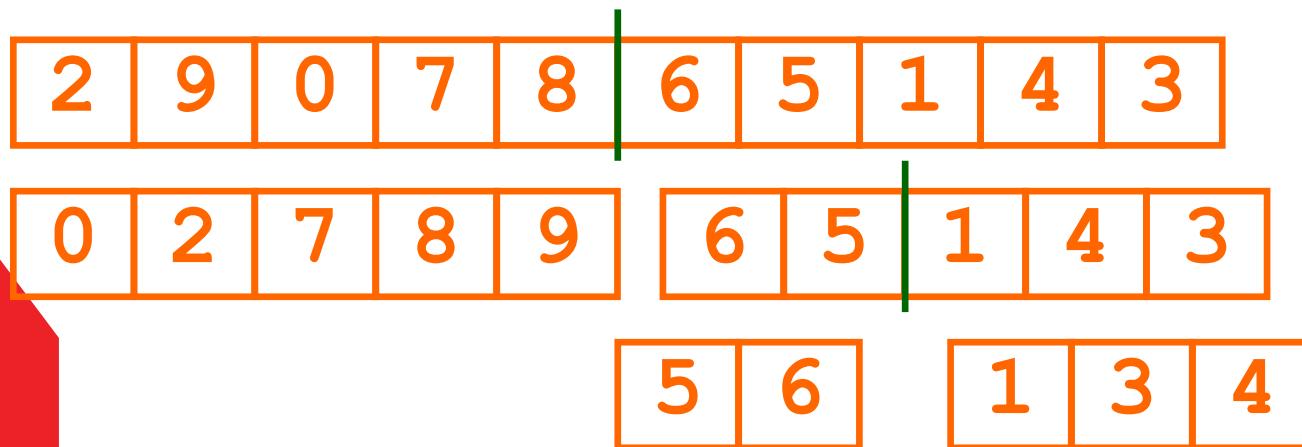
Merge Sort Animation



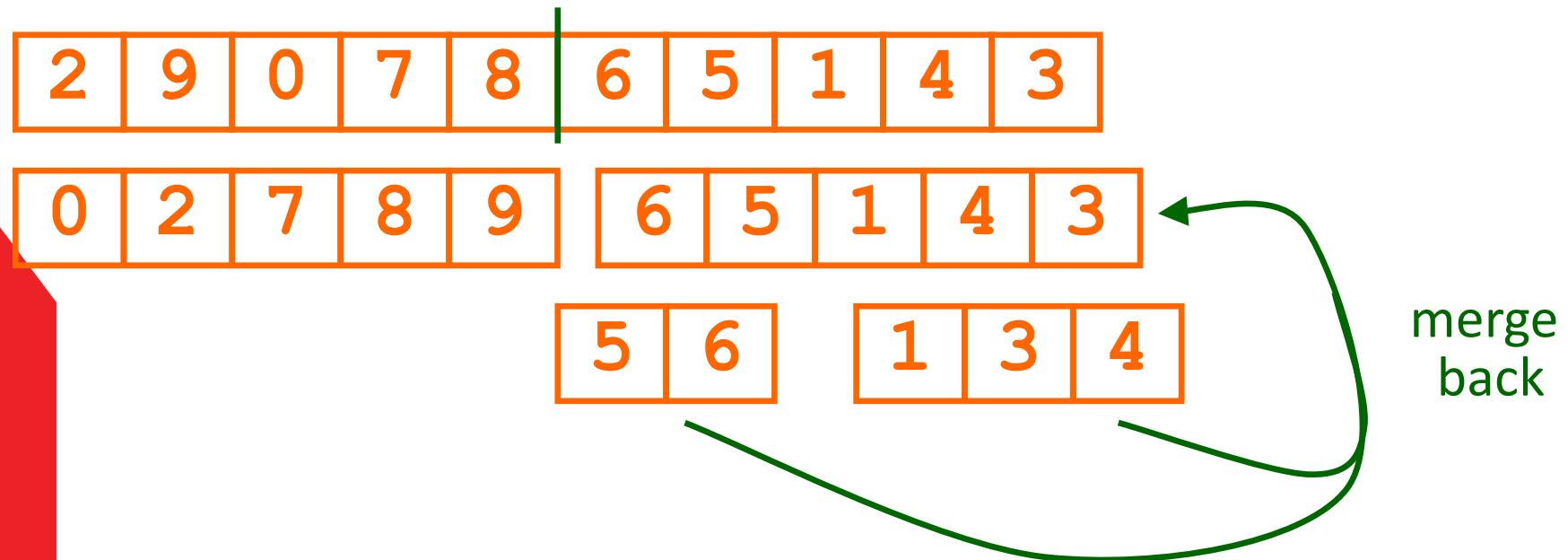
Merge Sort Animation



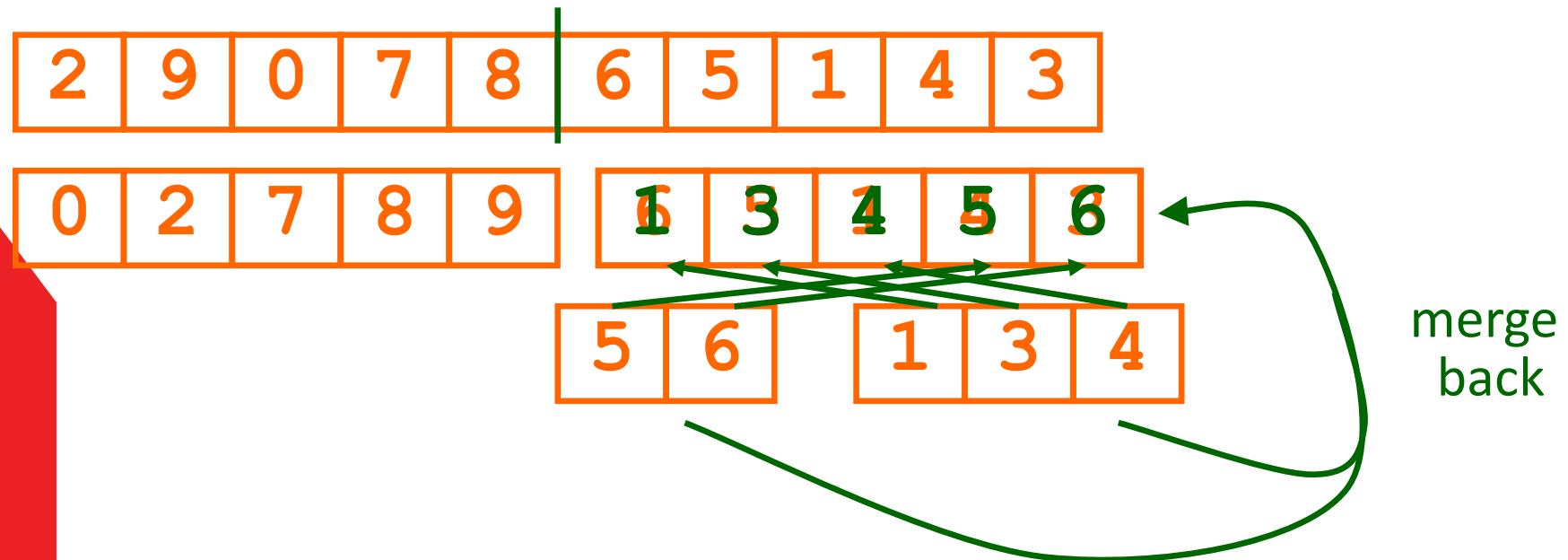
Merge Sort Animation



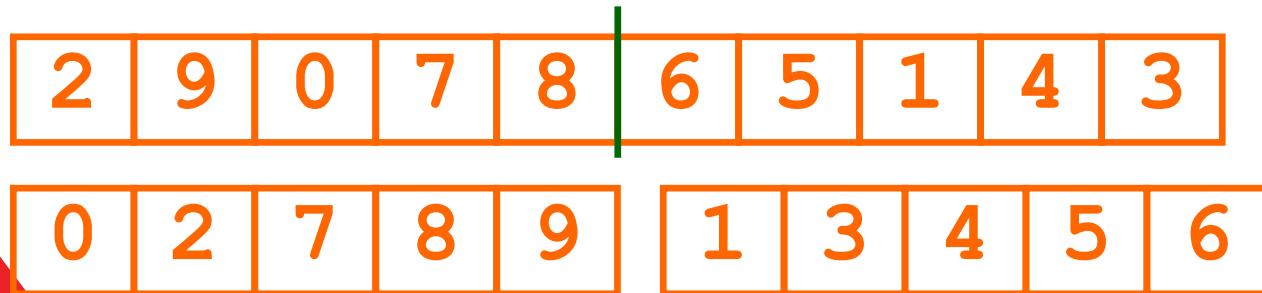
Merge Sort Animation



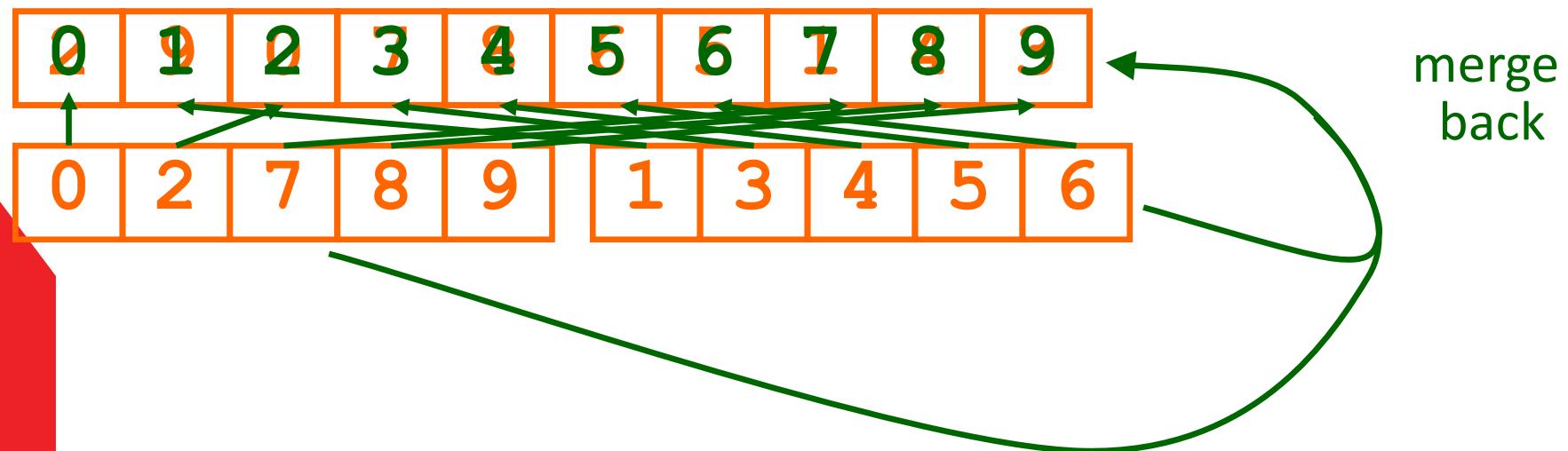
Merge Sort Animation



Merge Sort Animation



Merge Sort Animation



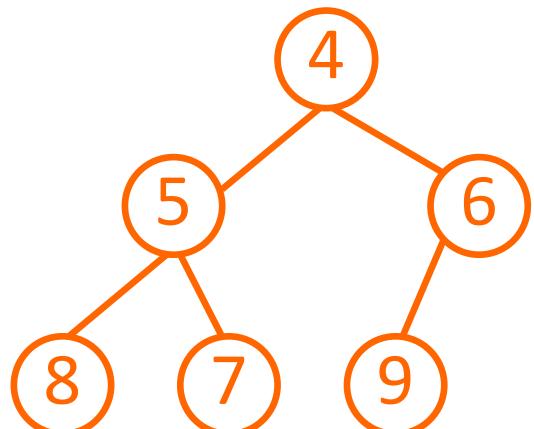
Merge Sort Animation



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.

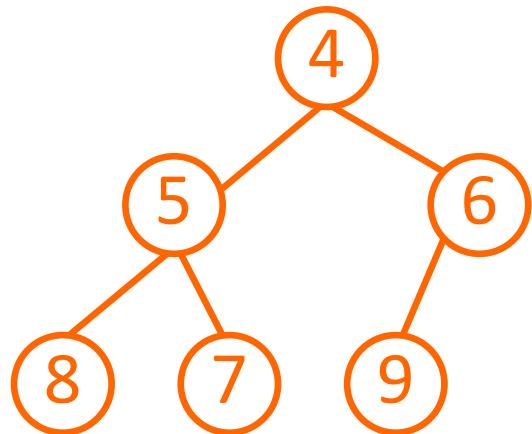


- A heap is a binary tree where the data in a node is guaranteed to be either less than (a min heap) or greater than (a max heap) all of the data below it.

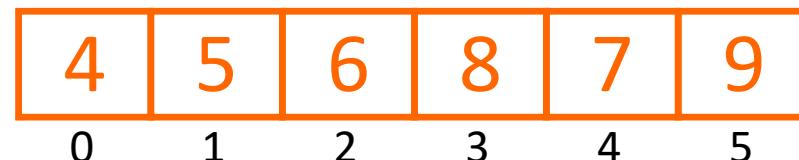
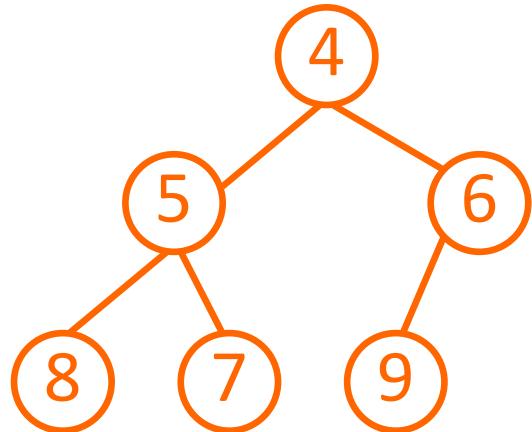
The Actual Heap

- 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

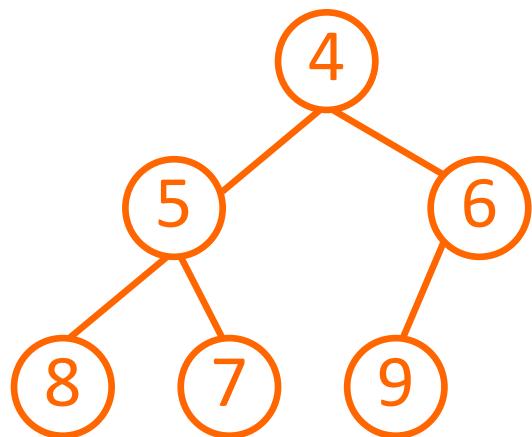


Abstract View vs Actual View

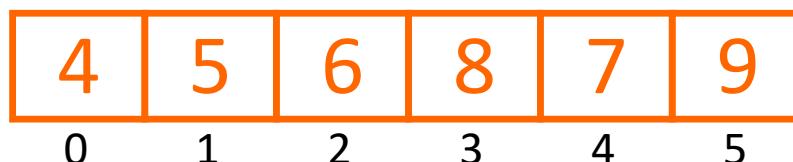


$\text{parentIndex} = (\text{childIndex} - 1) / 2$

Abstract View vs Actual View



`childIndex1 = parentIndex * 2 + 1`



`childIndex2 = parentIndex * 2 + 2`

Heap Sort Algorithm

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

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

Heap Insert Animation

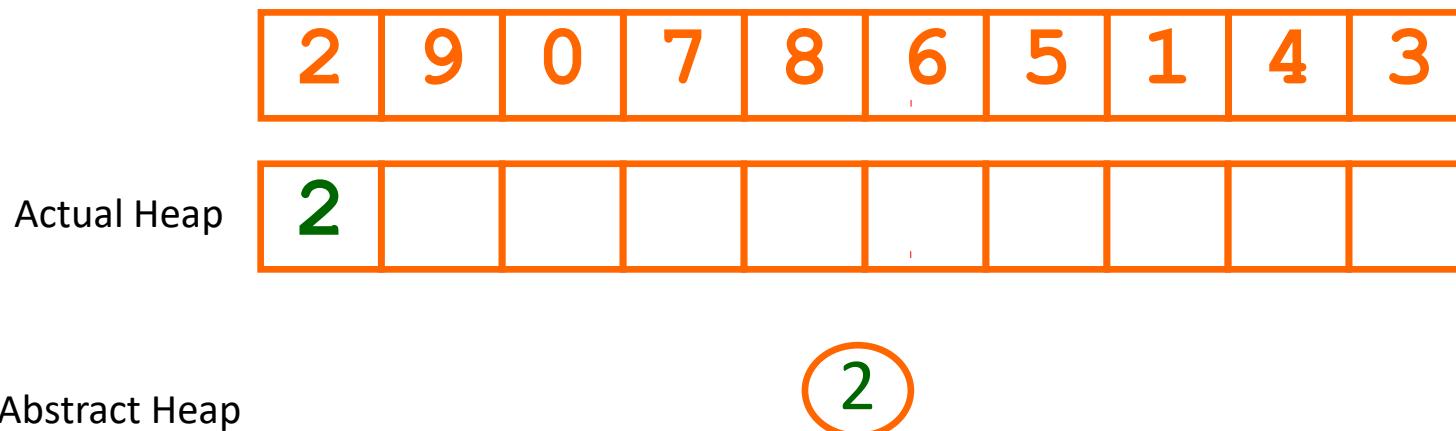
2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

Actual Heap

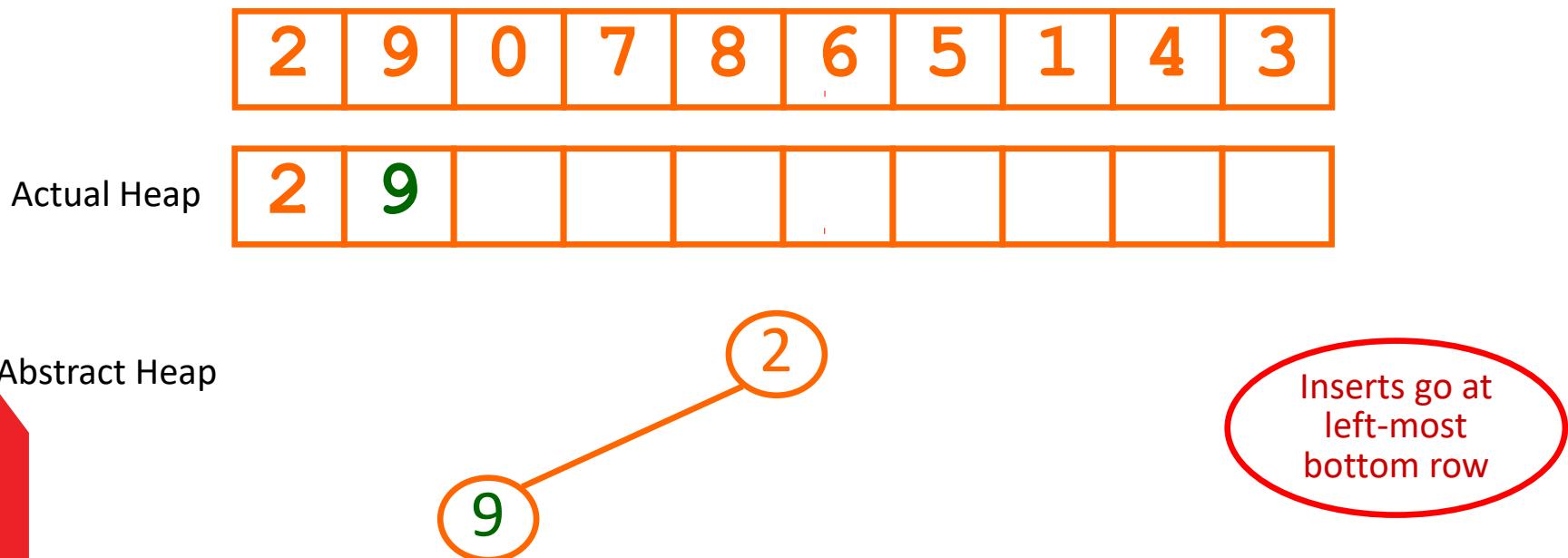
--	--	--	--	--	--	--	--	--	--

Abstract Heap

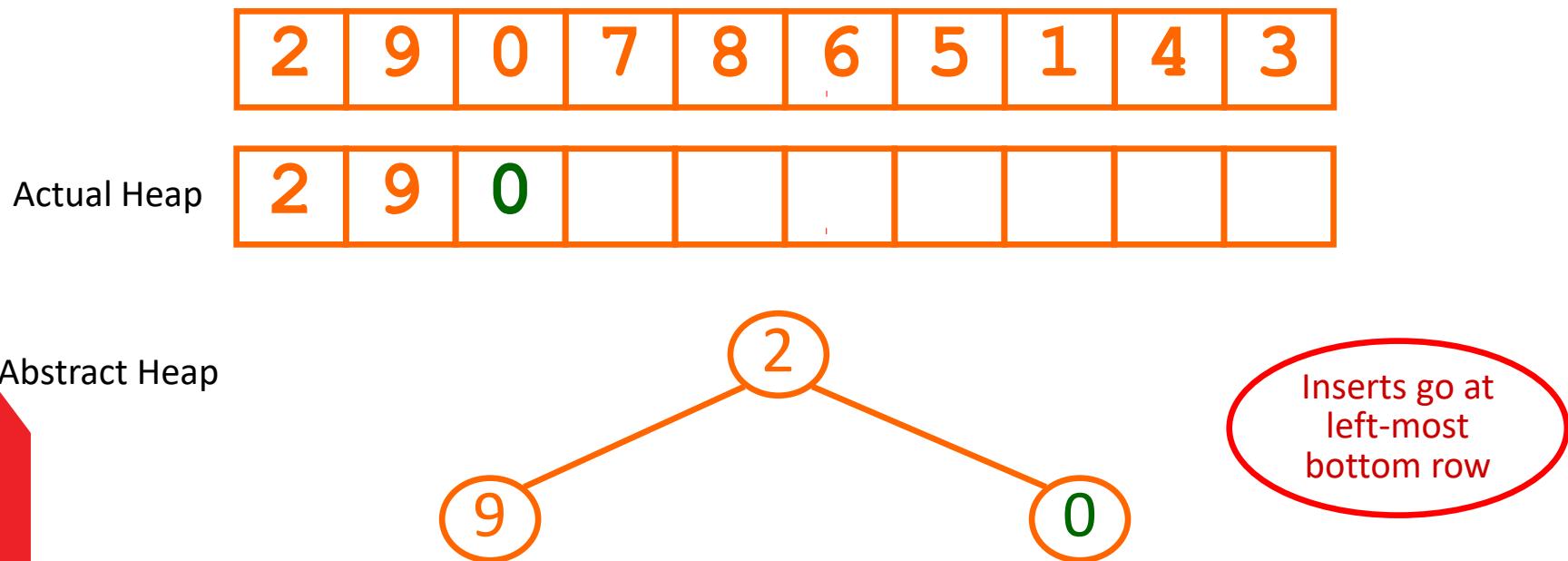
Heap Insert Animation



Heap Insert Animation



Heap Insert Animation

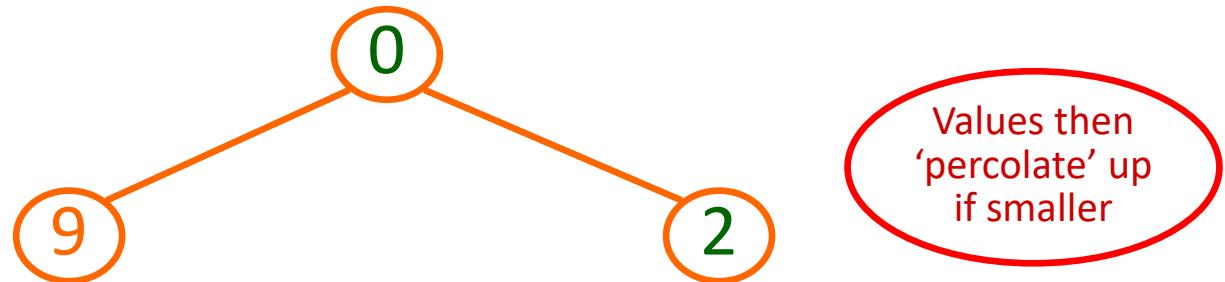


Heap Insert Animation

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

Actual Heap

Abstract Heap



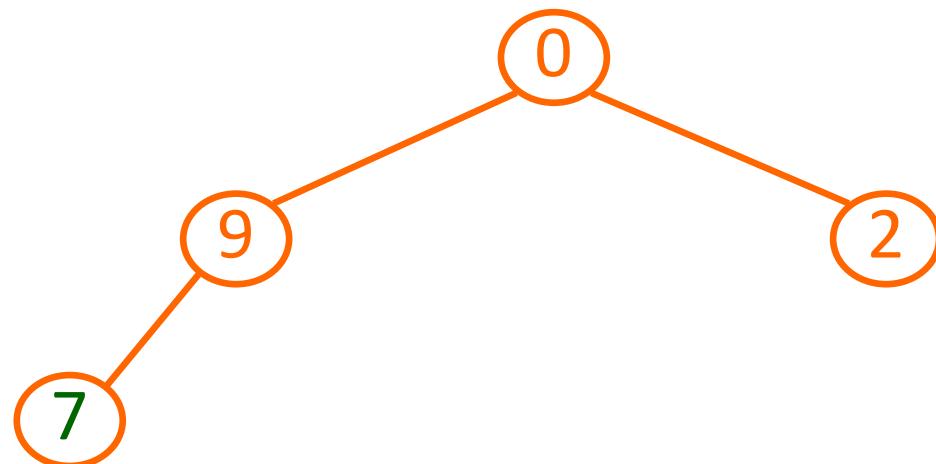
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

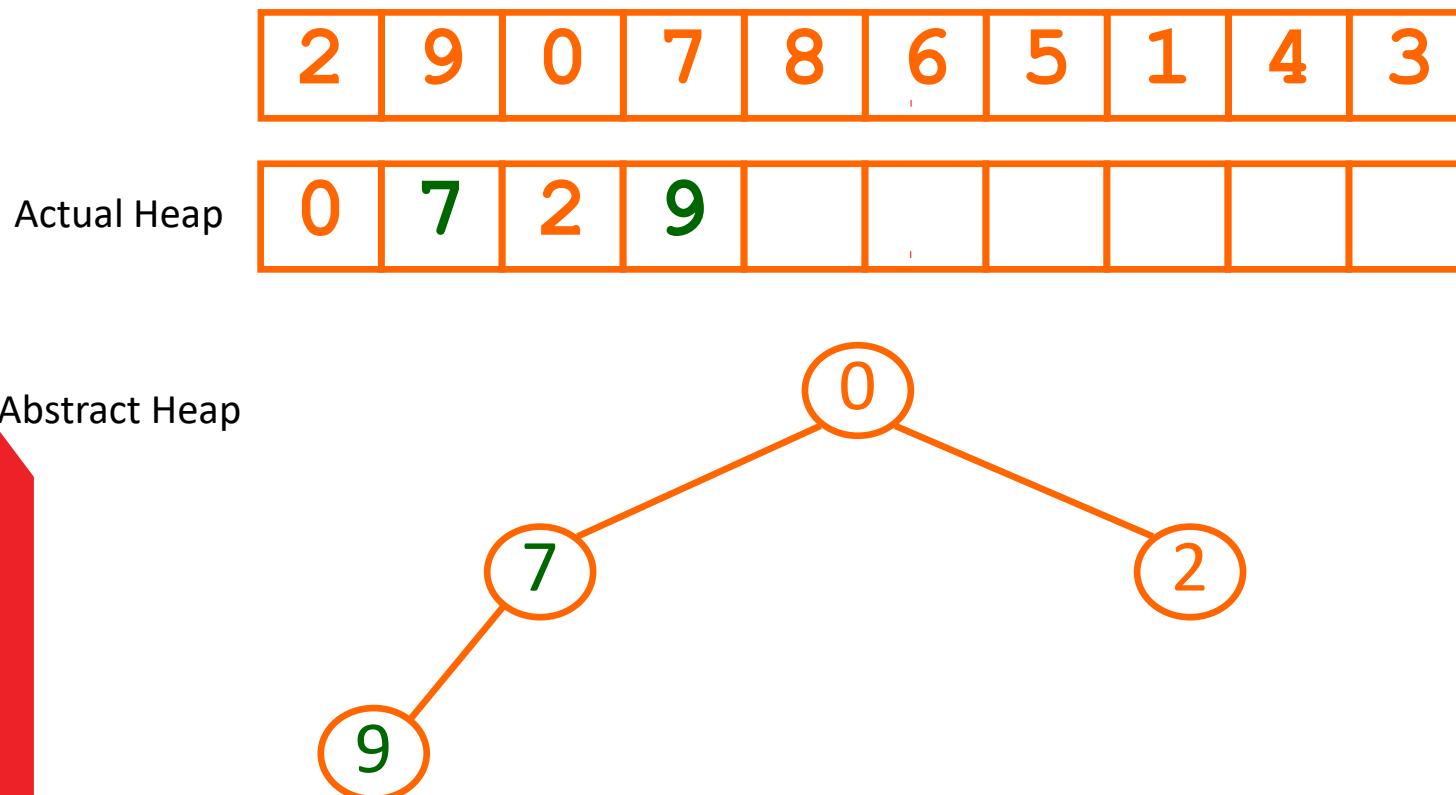
Actual Heap

0	9	2	7						
---	---	---	---	--	--	--	--	--	--

Abstract Heap



Heap Insert Animation



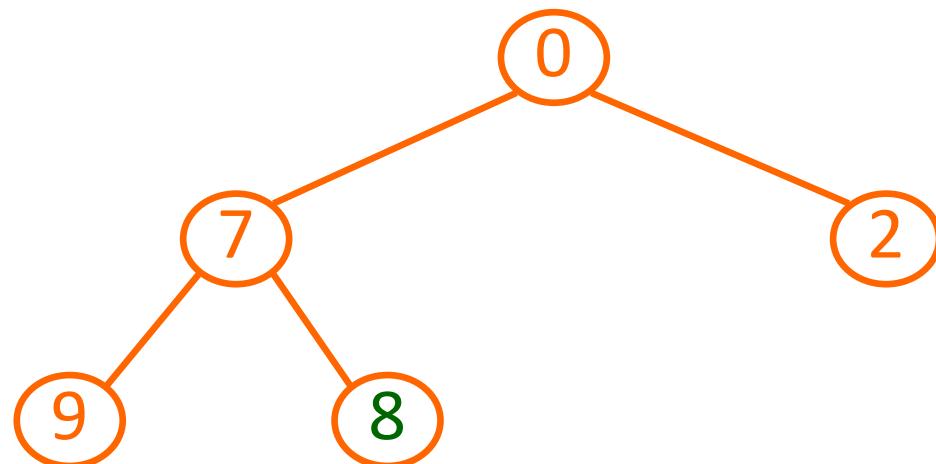
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

Actual Heap

0	7	2	9	8					
---	---	---	---	---	--	--	--	--	--

Abstract Heap



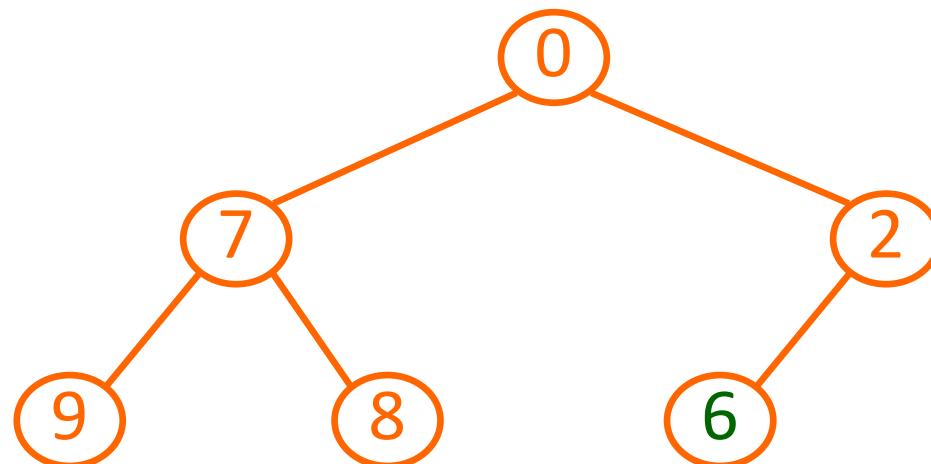
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

Actual Heap

0	7	2	9	8	6				
---	---	---	---	---	---	--	--	--	--

Abstract Heap



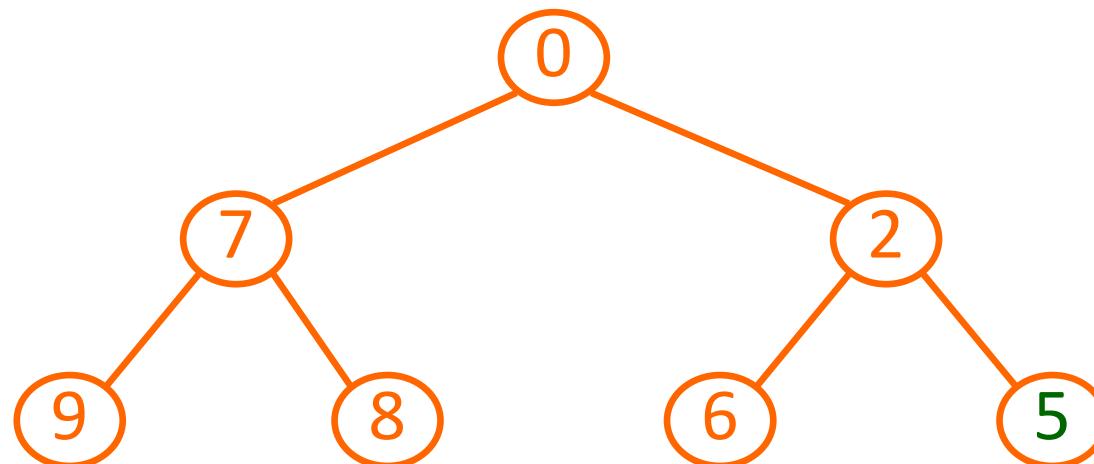
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

Actual Heap

0	7	2	9	8	6	5			
---	---	---	---	---	---	---	--	--	--

Abstract Heap



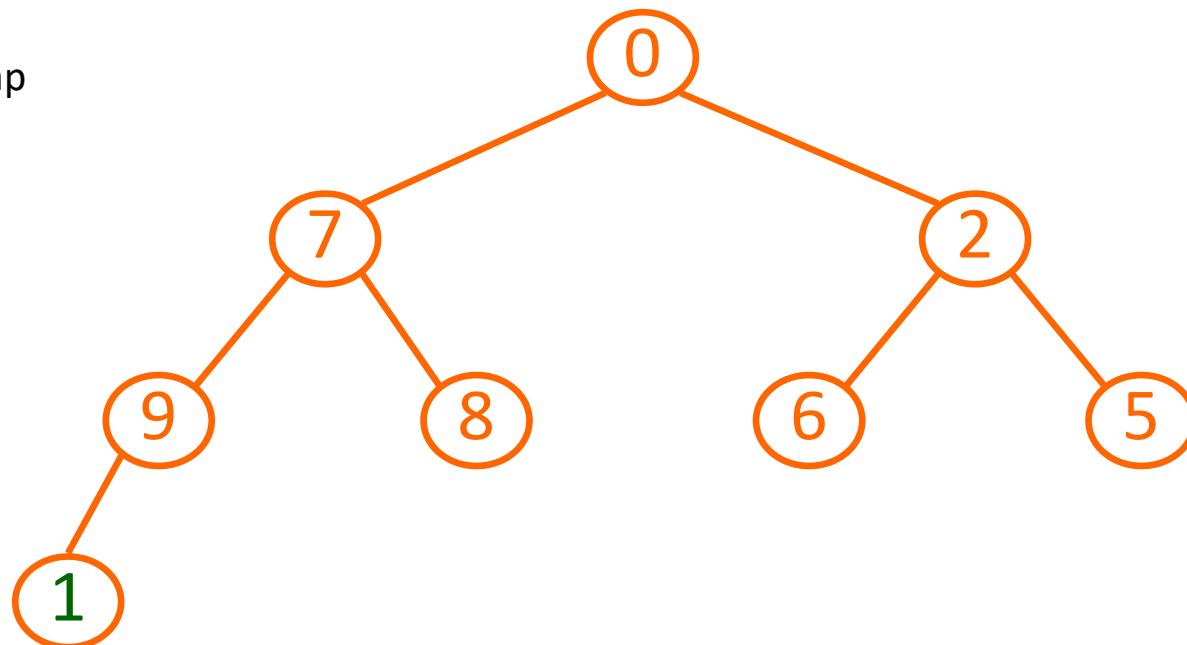
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

Actual Heap

0	7	2	9	8	6	5	1		
---	---	---	---	---	---	---	---	--	--

Abstract Heap



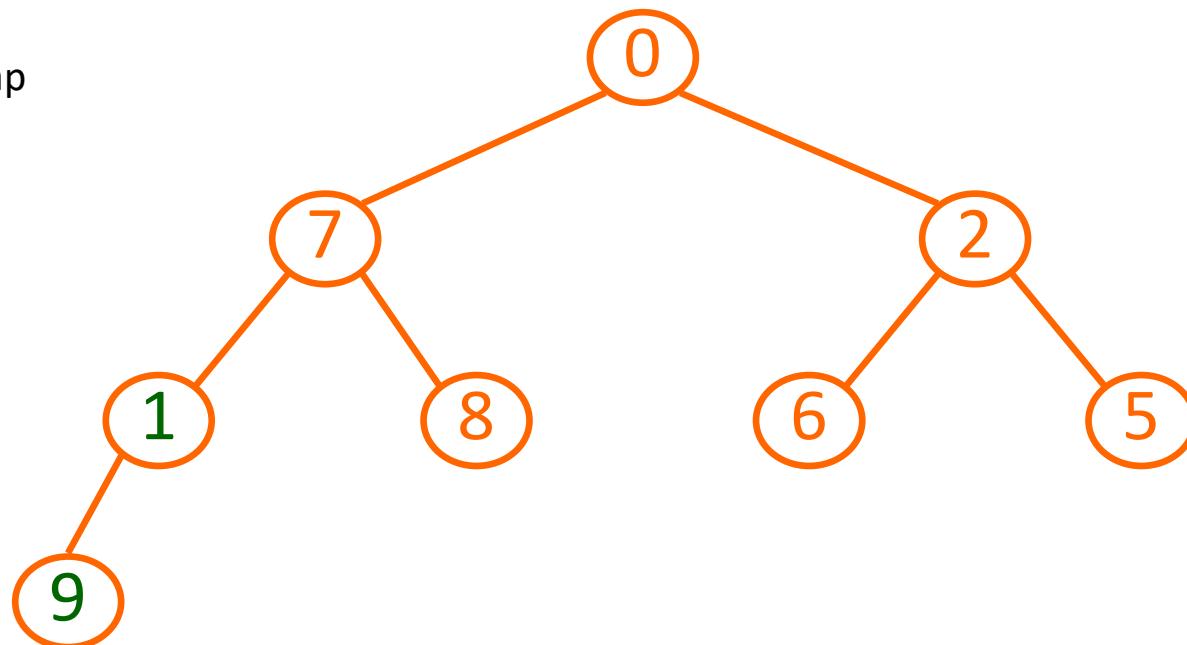
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

Actual Heap

0	7	2	1	8	6	5	9		
---	---	---	---	---	---	---	---	--	--

Abstract Heap



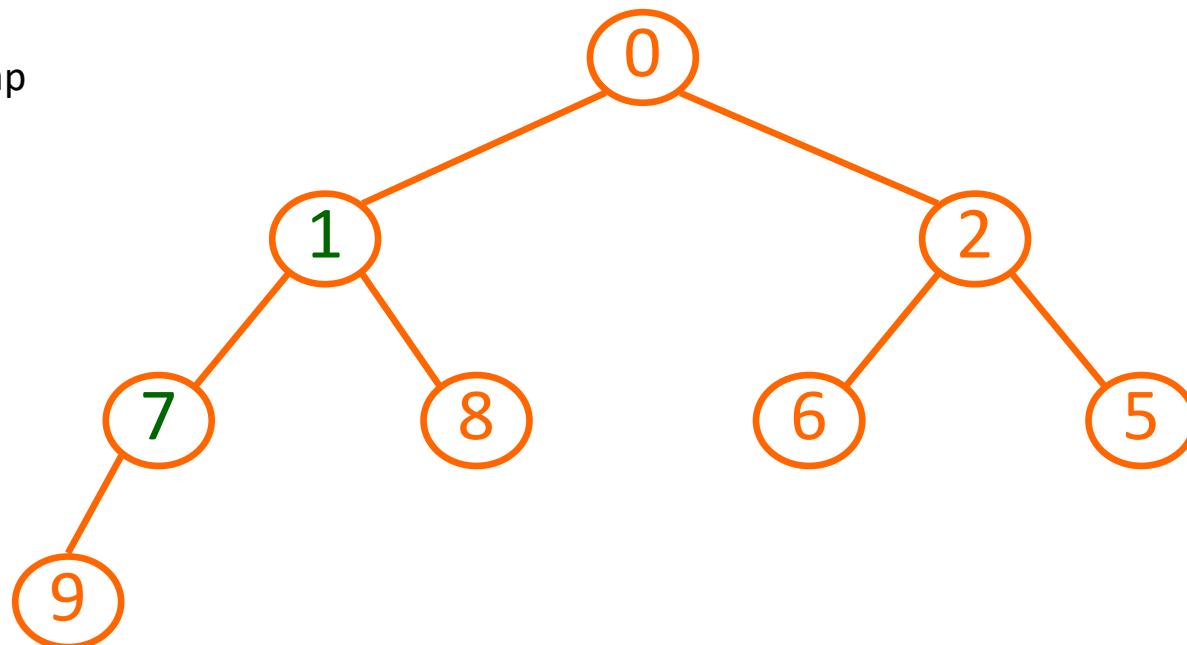
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

Actual Heap

0	1	2	7	8	6	5	9		
---	---	---	---	---	---	---	---	--	--

Abstract Heap



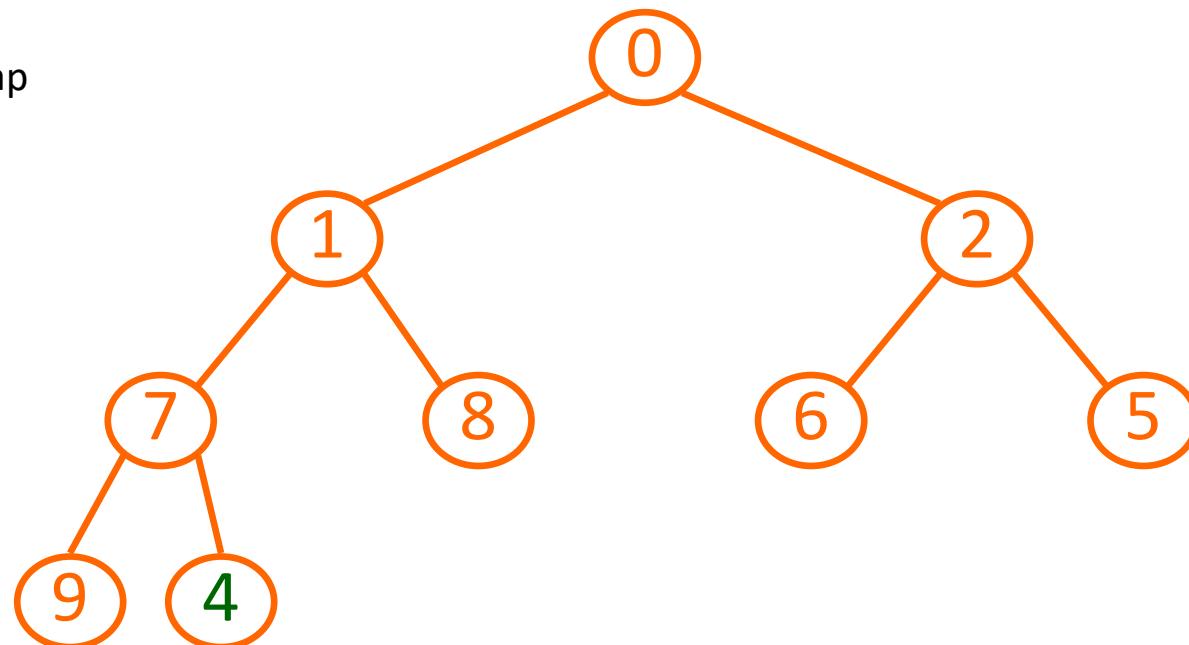
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

Actual Heap

0	1	2	7	8	6	5	9	4	
---	---	---	---	---	---	---	---	---	--

Abstract Heap



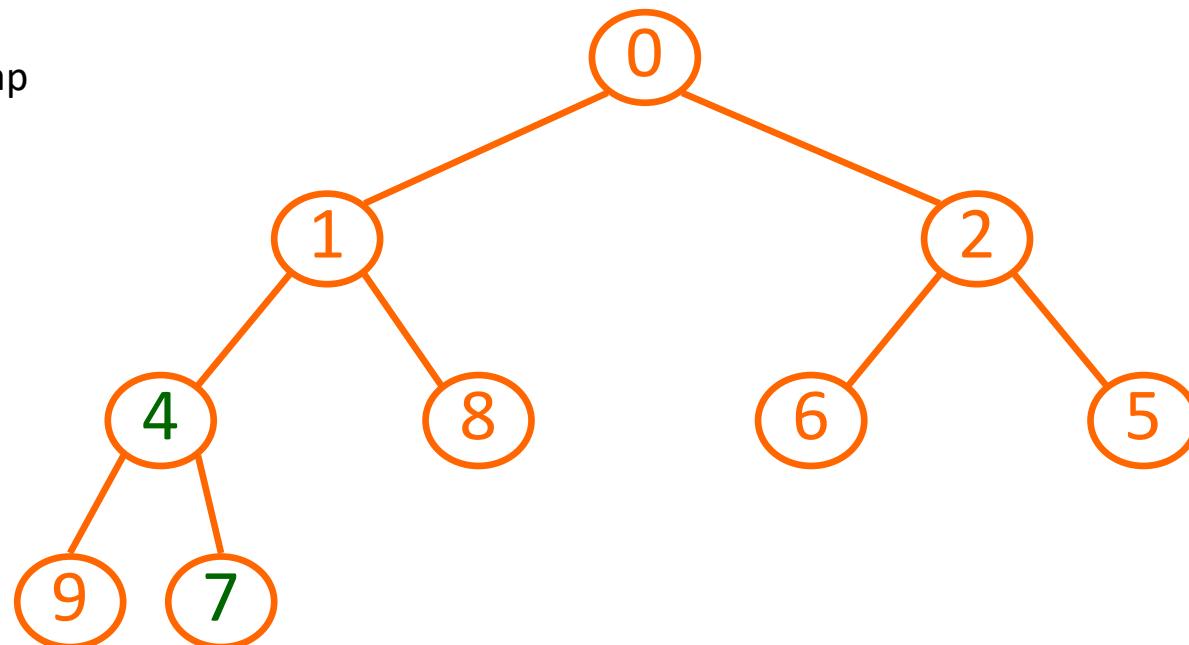
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

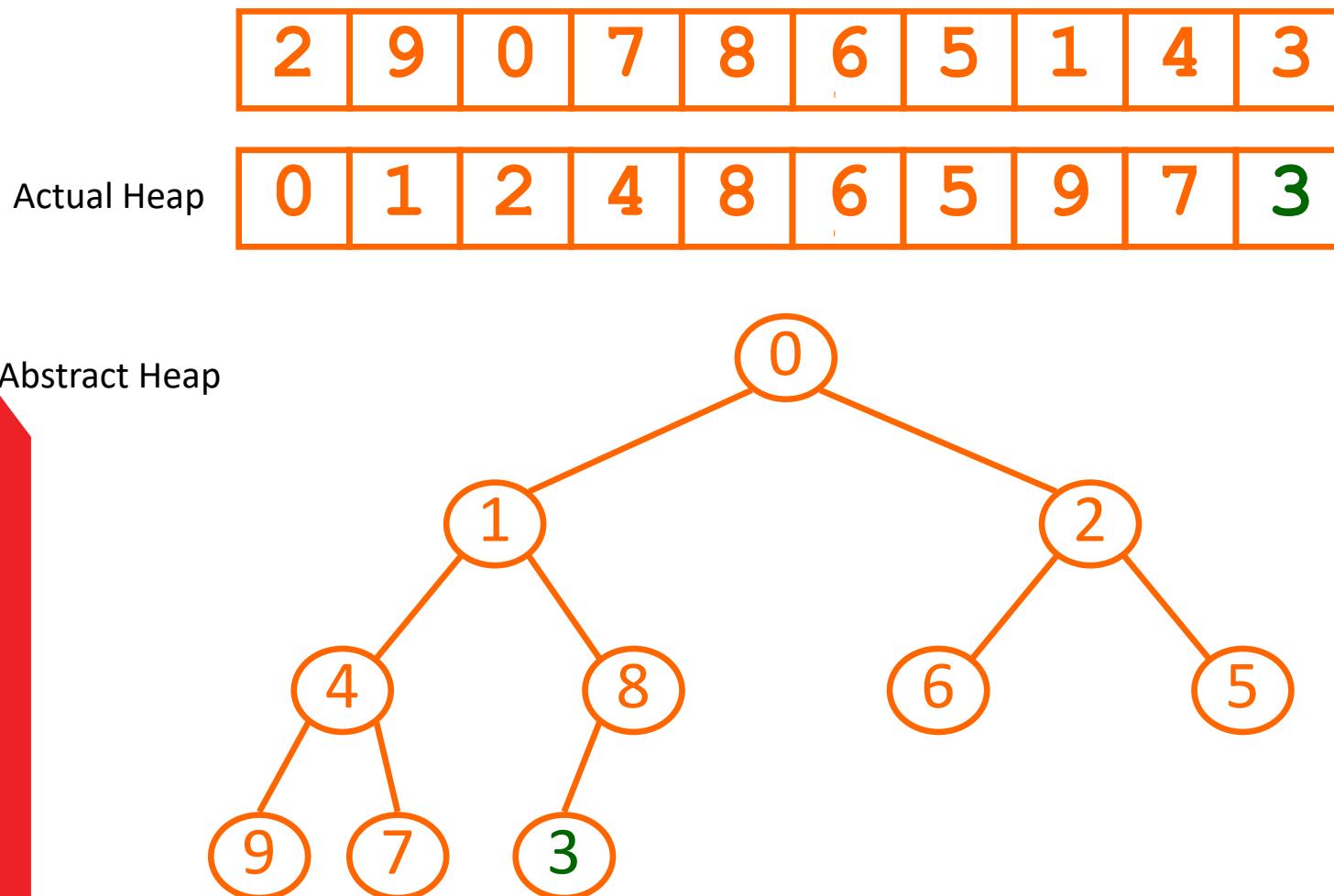
Actual Heap

0	1	2	4	8	6	5	9	7	
---	---	---	---	---	---	---	---	---	--

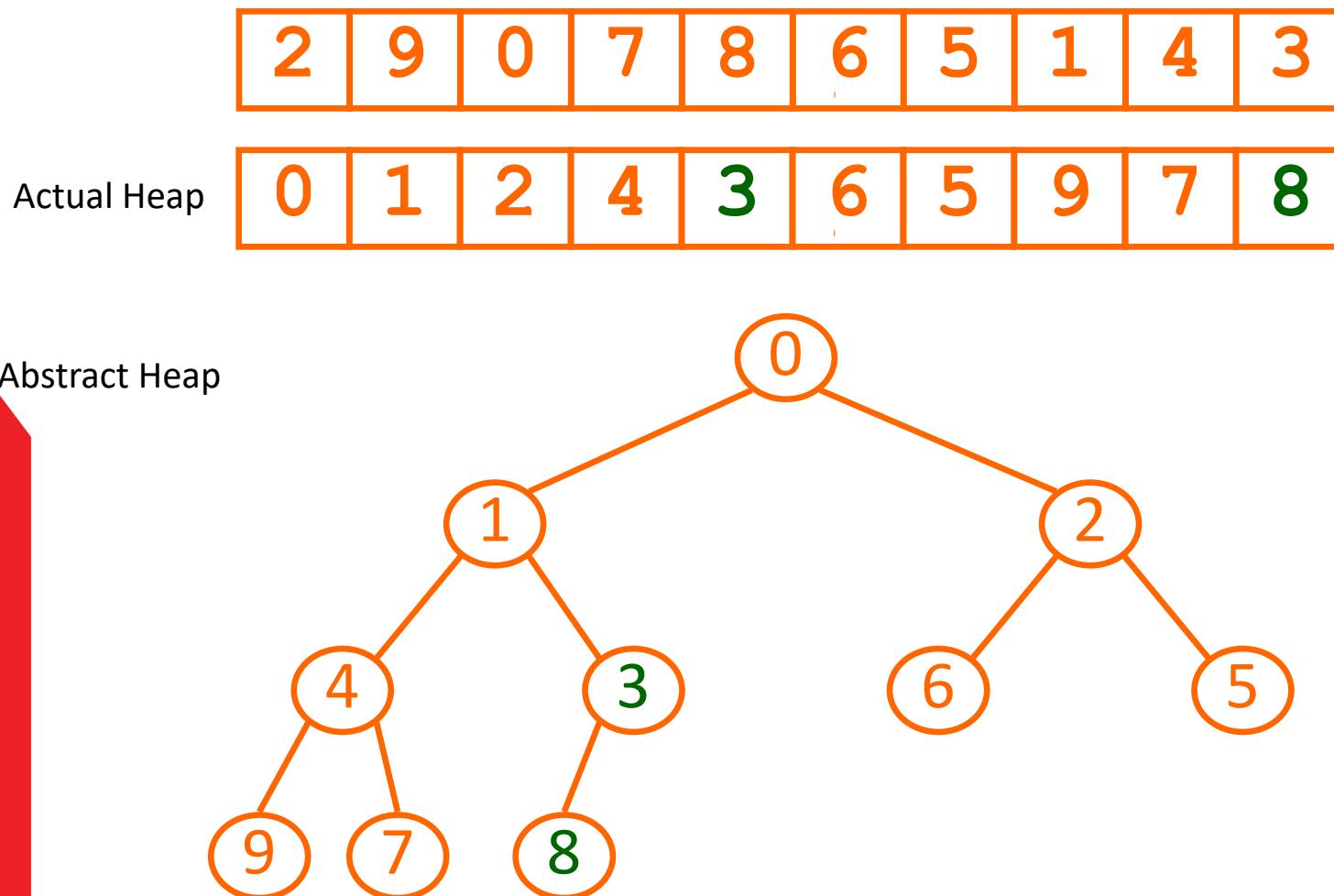
Abstract Heap



Heap Insert Animation



Heap Insert Animation



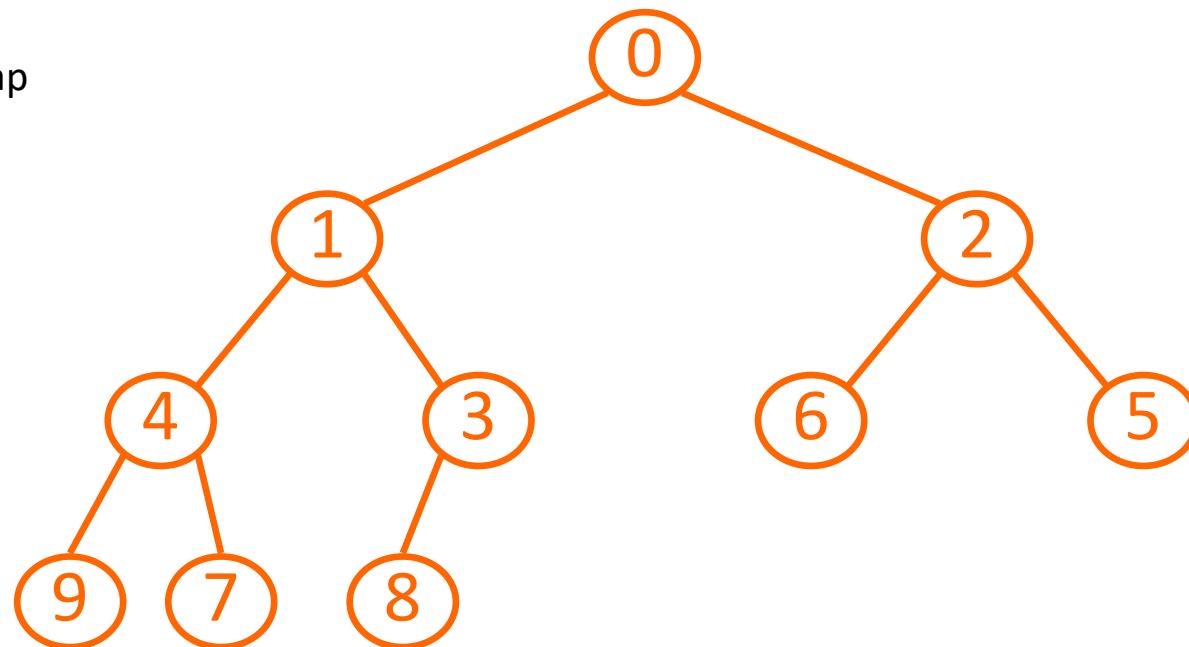
Heap Insert Animation

2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

Actual Heap

0	1	2	4	3	6	5	9	7	8
---	---	---	---	---	---	---	---	---	---

Abstract Heap



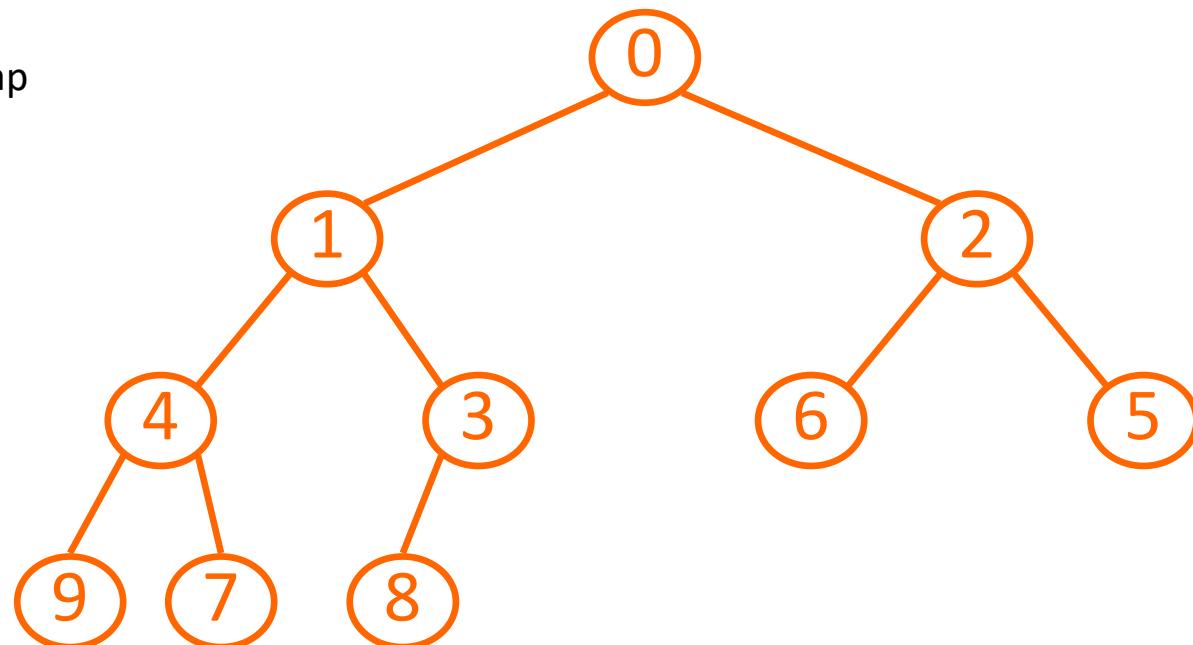
Heap Delete Animation

2	9	0	7	8	6	5	1	4	3
---	---	---	---	---	---	---	---	---	---

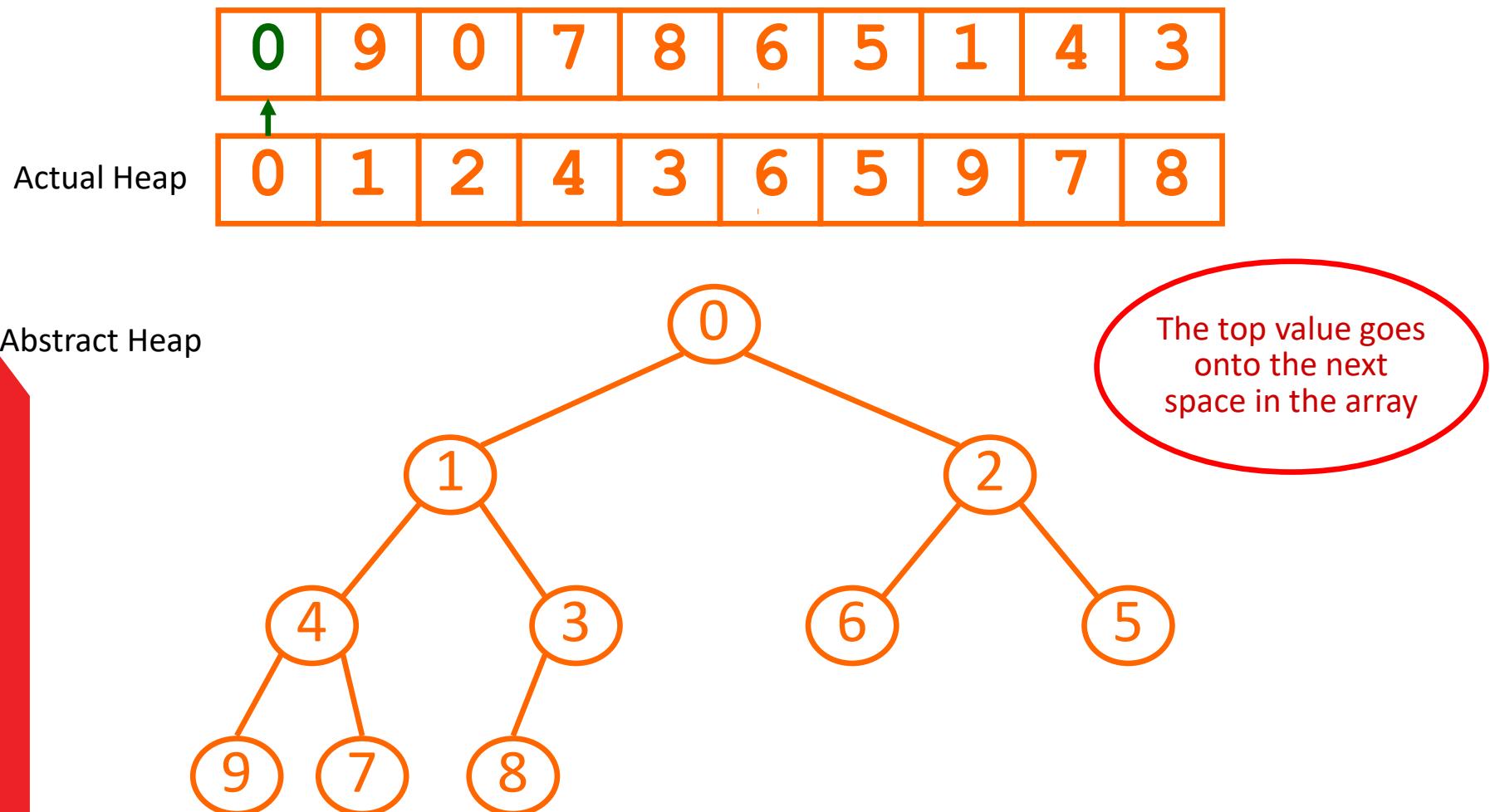
Actual Heap

0	1	2	4	3	6	5	9	7	8
---	---	---	---	---	---	---	---	---	---

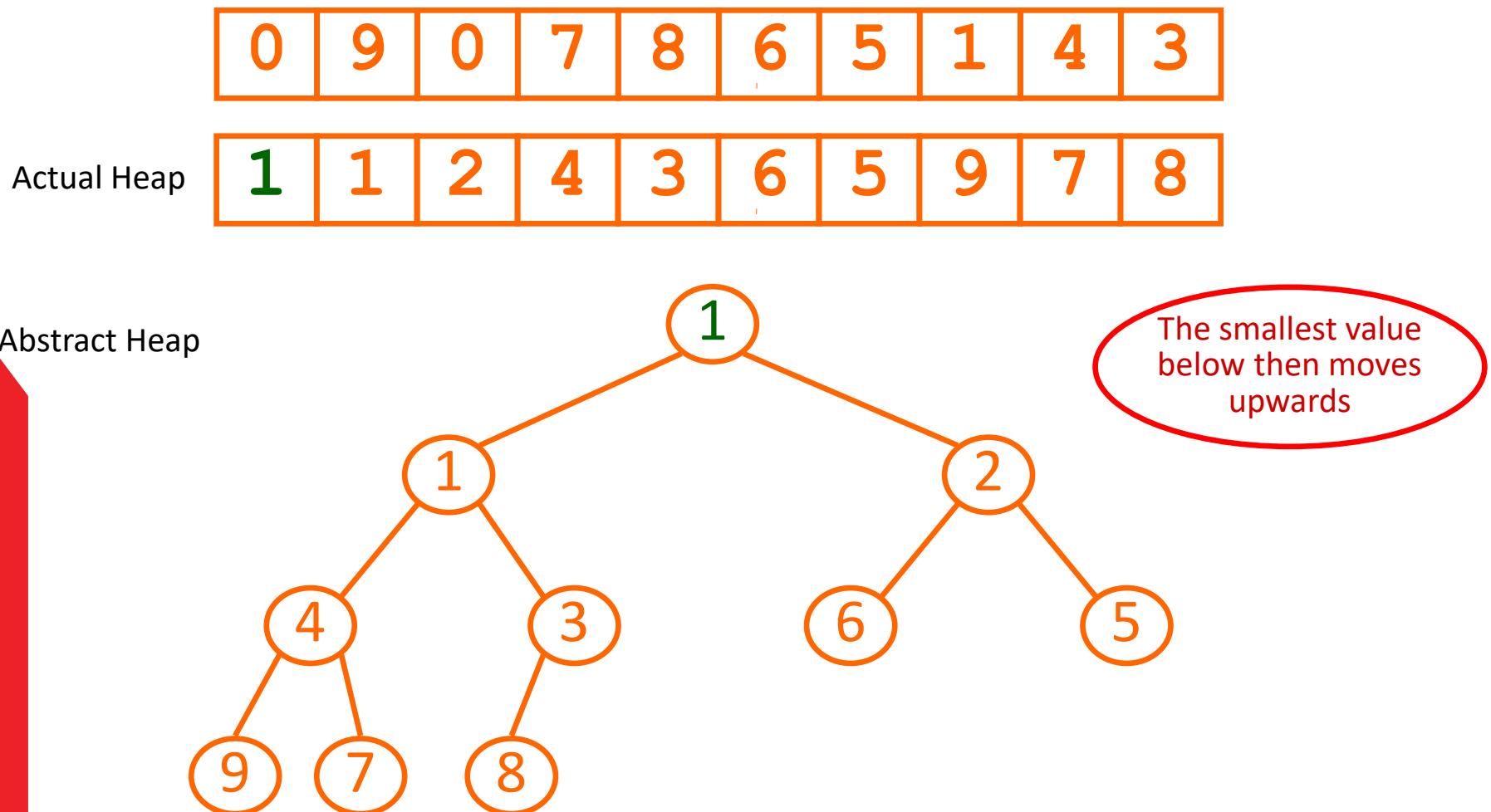
Abstract Heap



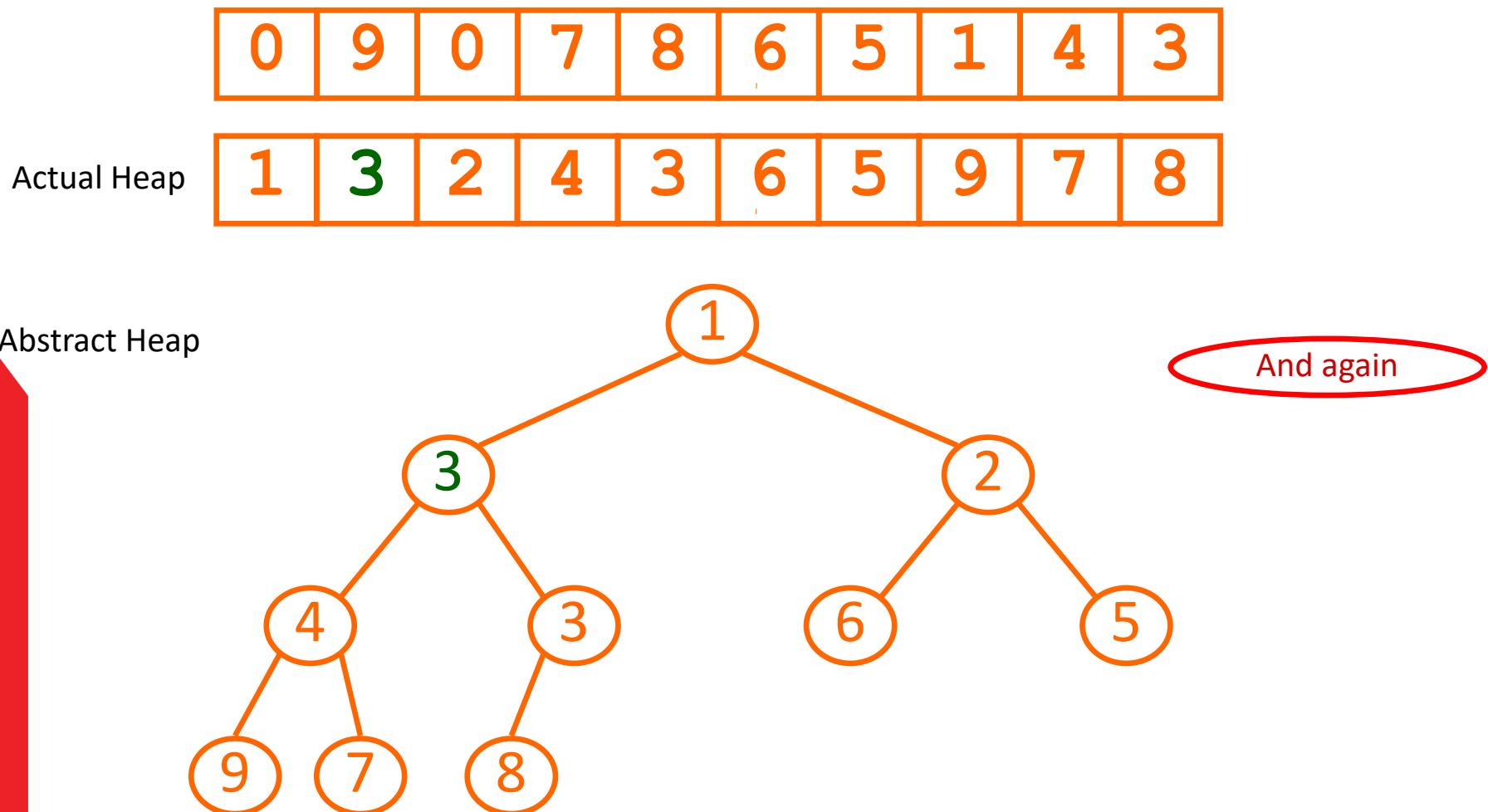
Heap Delete Animation



Heap Delete Animation



Heap Delete Animation

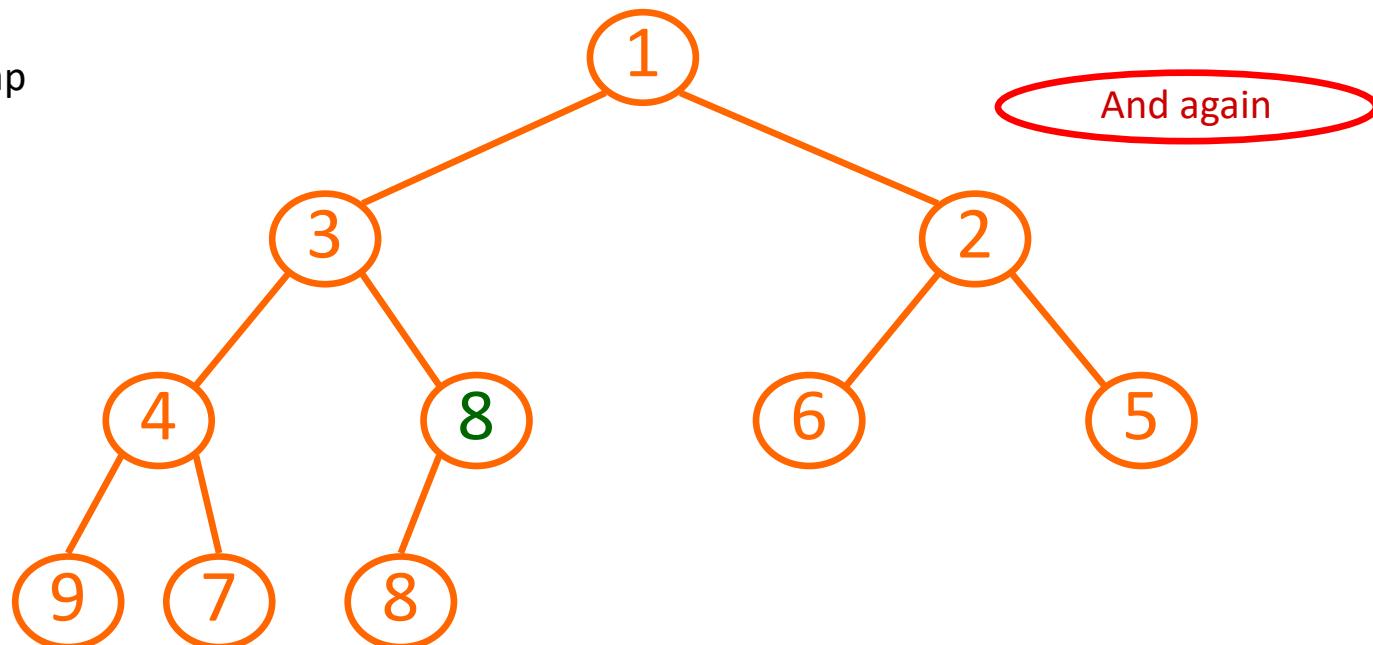


Heap Delete Animation

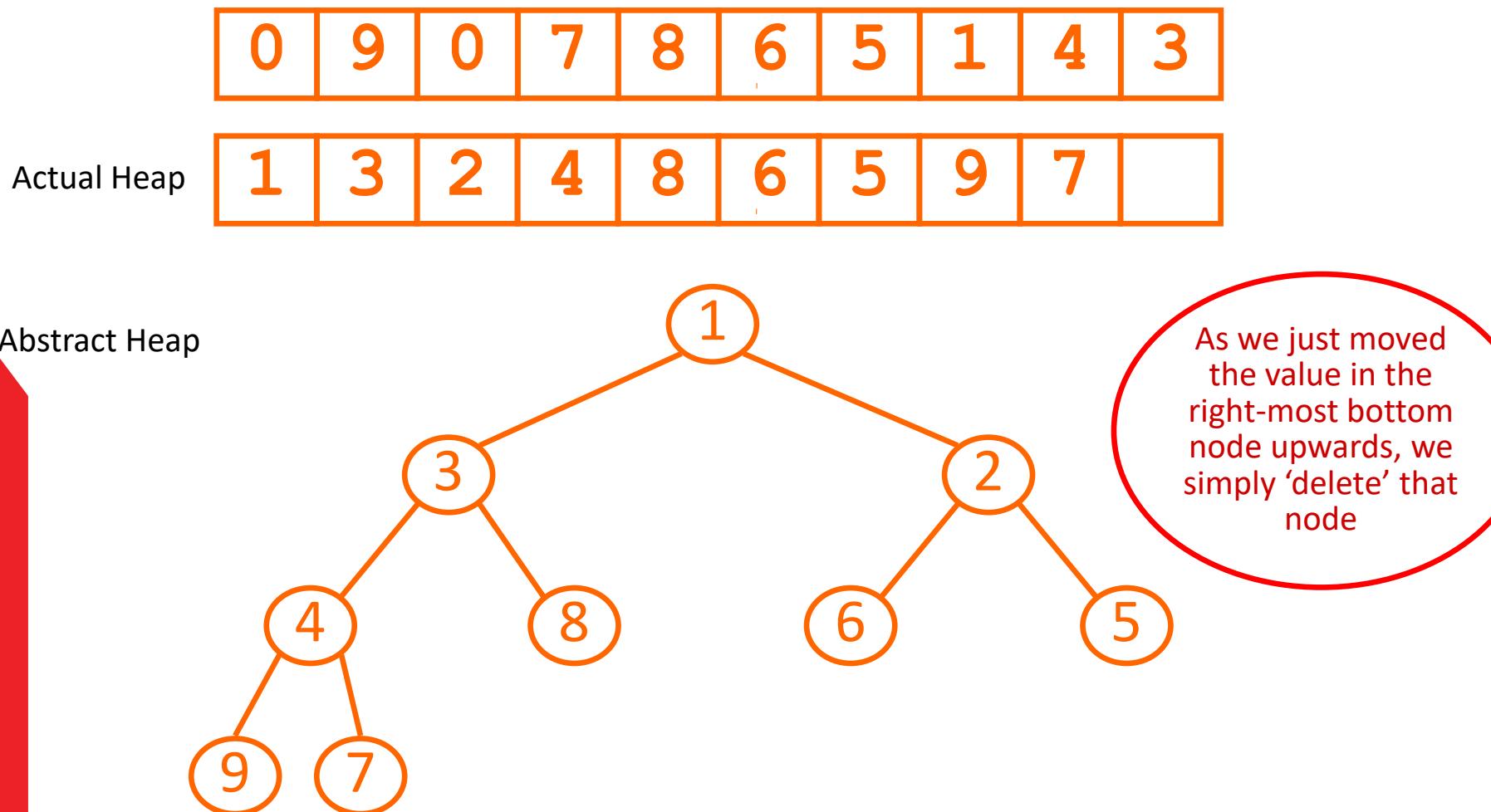
0	9	0	7	8	6	5	1	4	3
1	3	2	4	8	6	5	9	7	8

Actual Heap

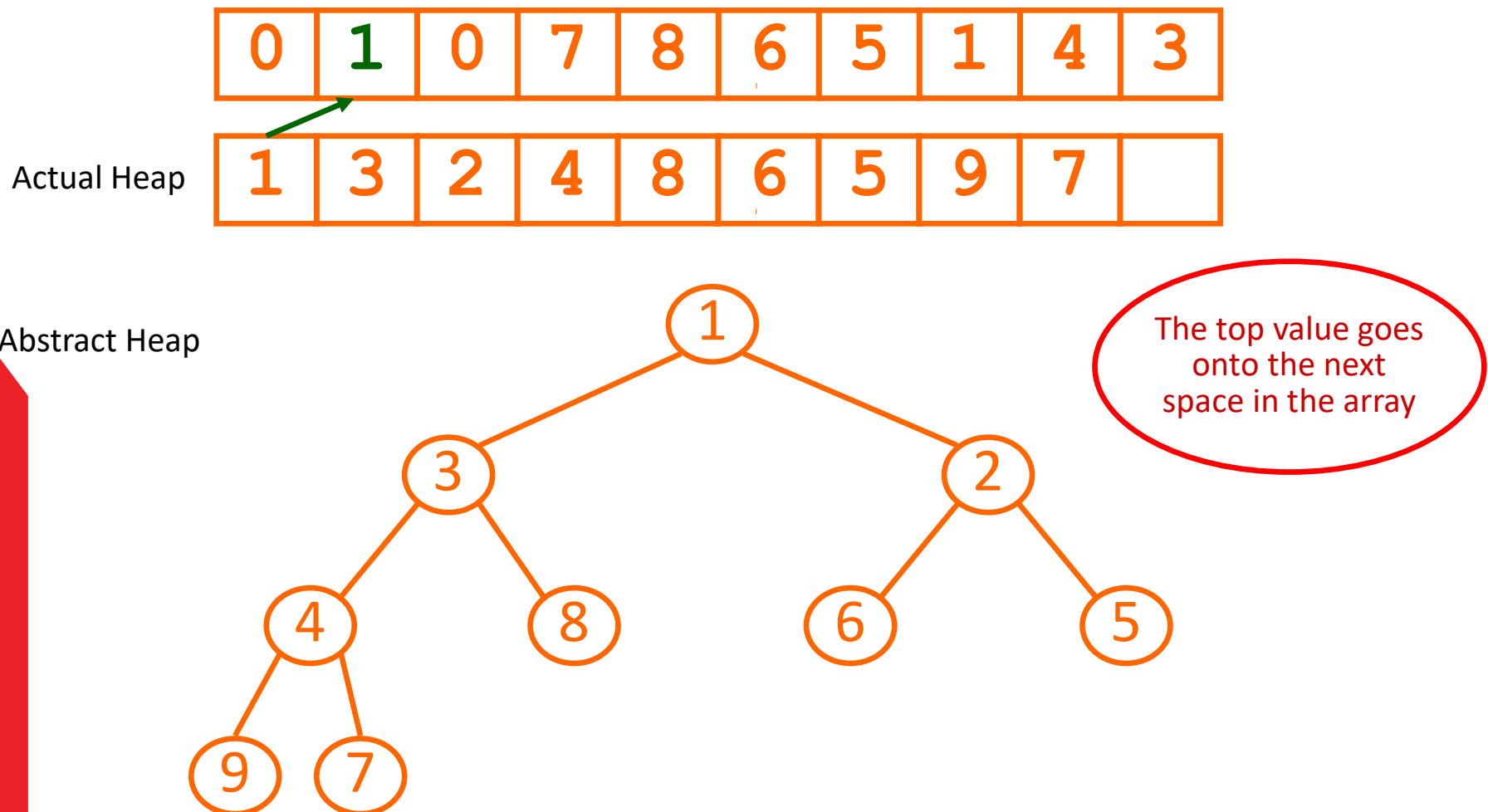
Abstract Heap



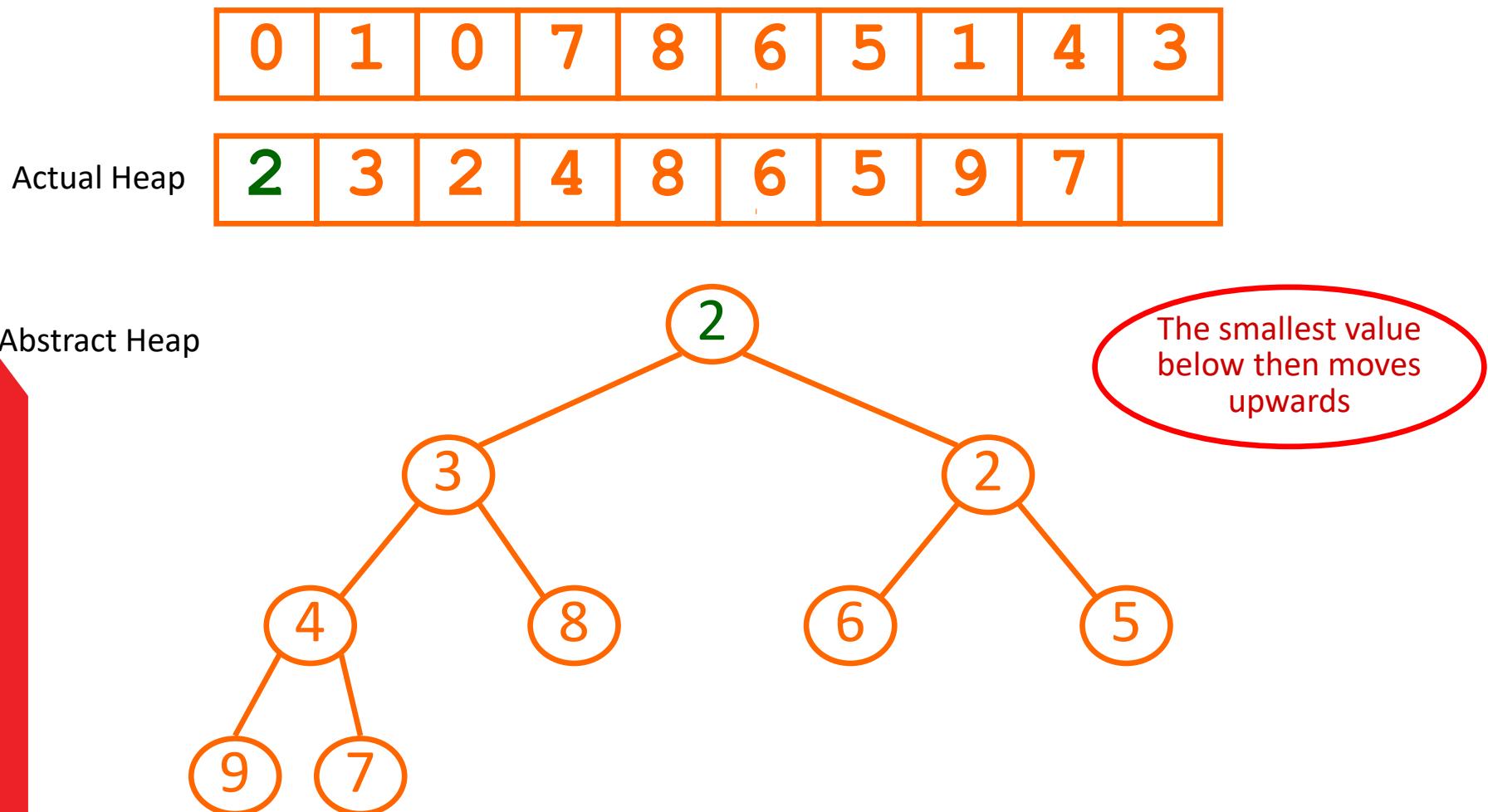
Heap Delete Animation



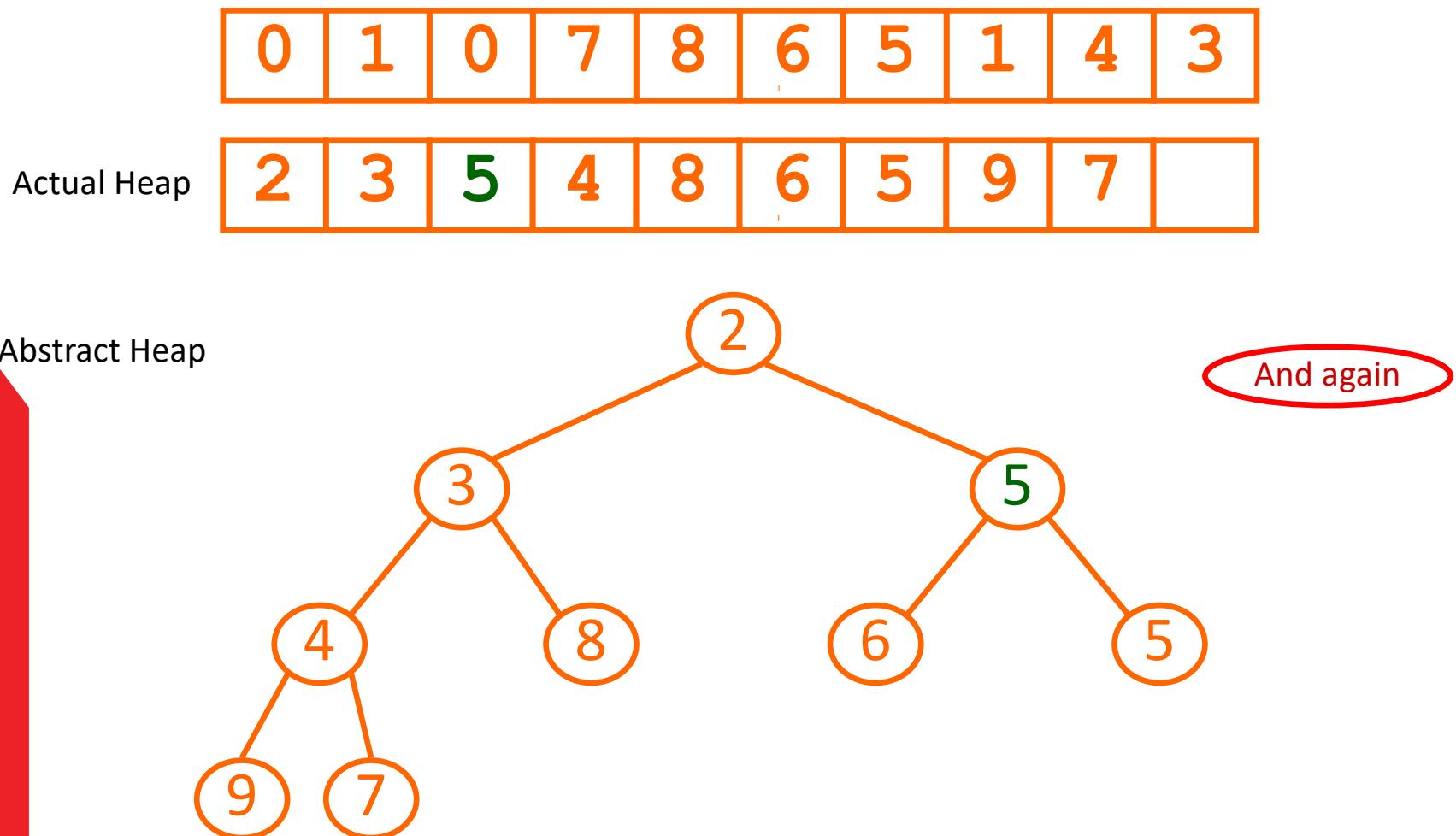
Heap Delete Animation



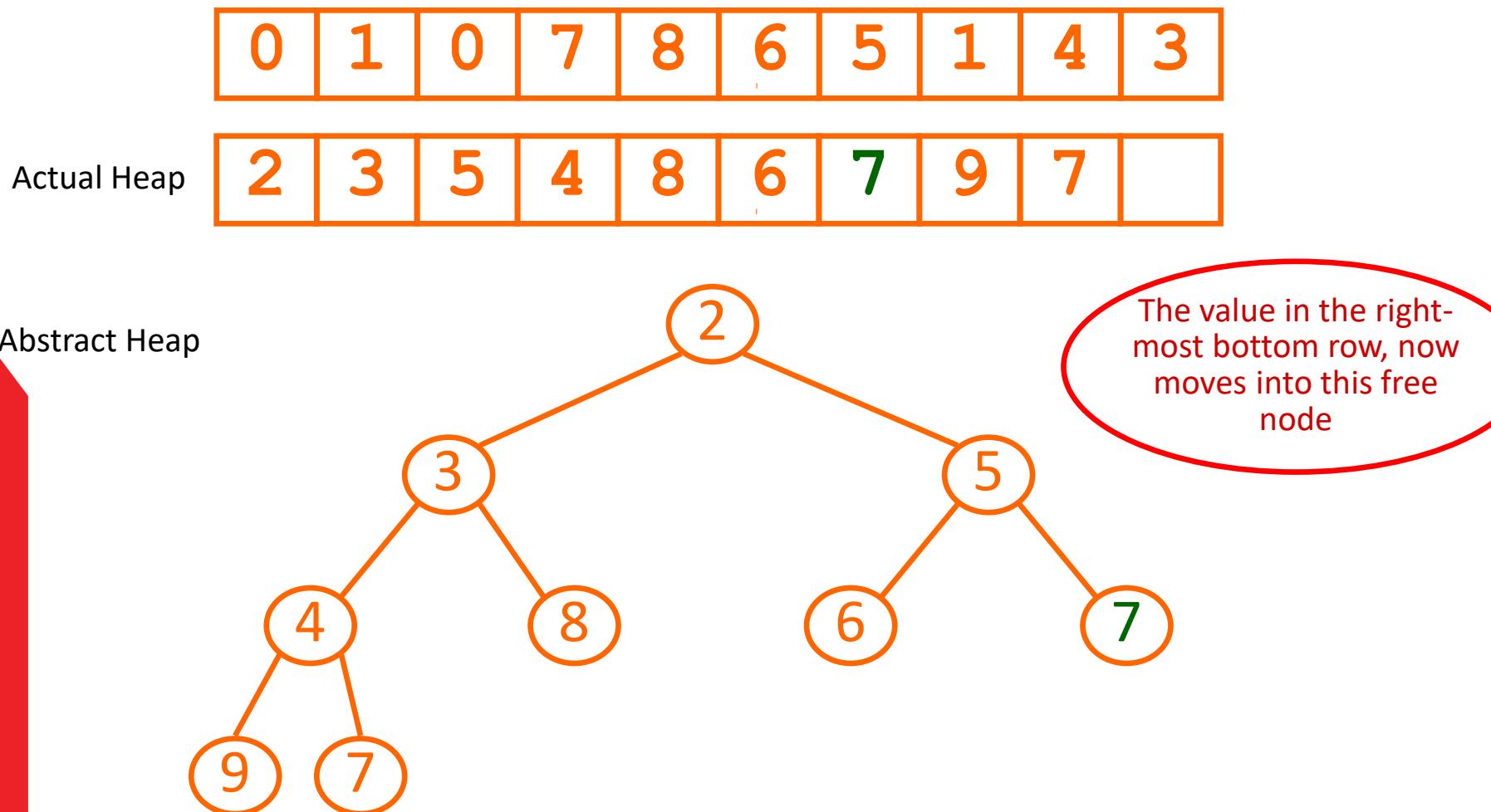
Heap Delete Animation



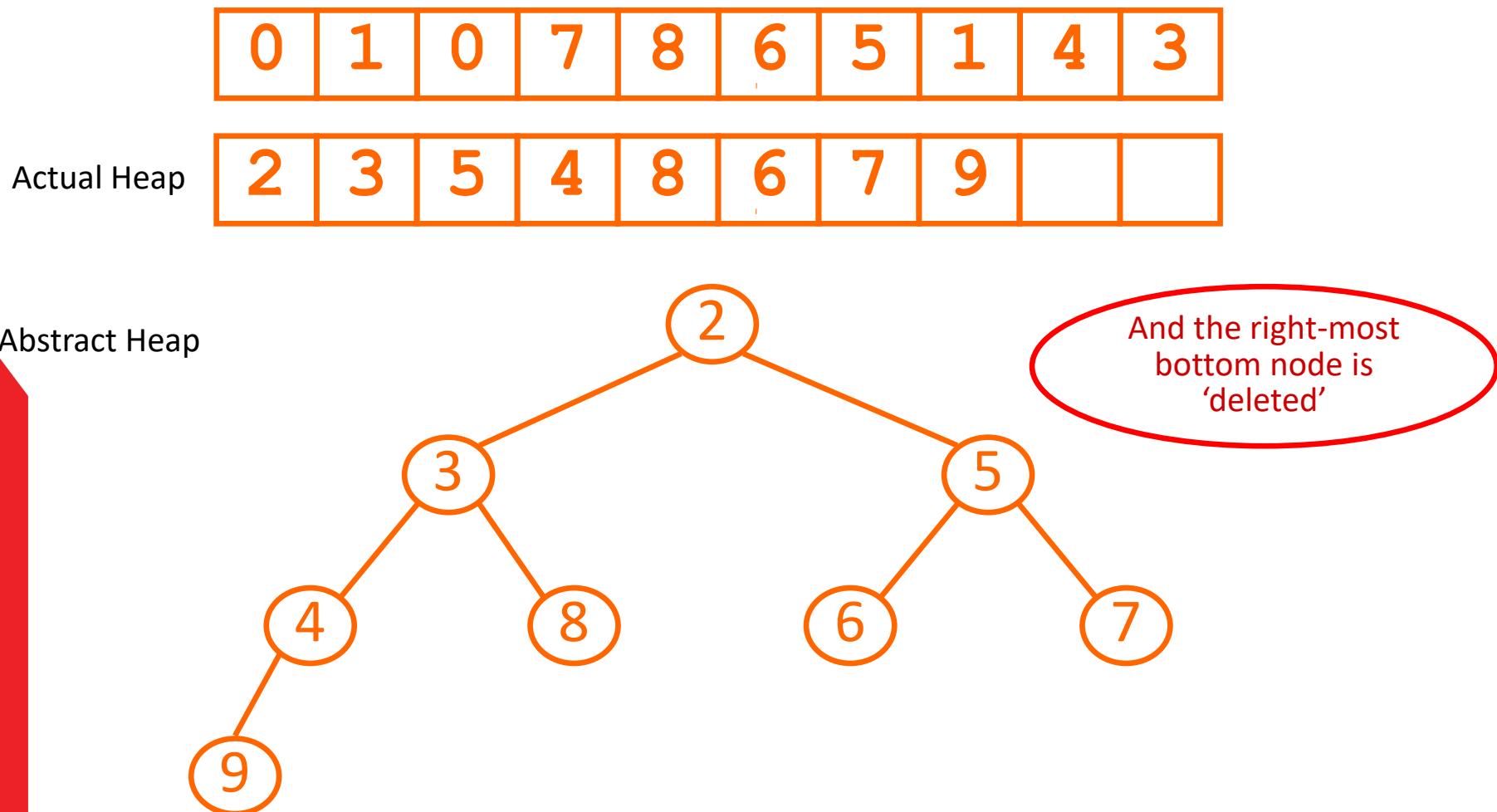
Heap Delete Animation



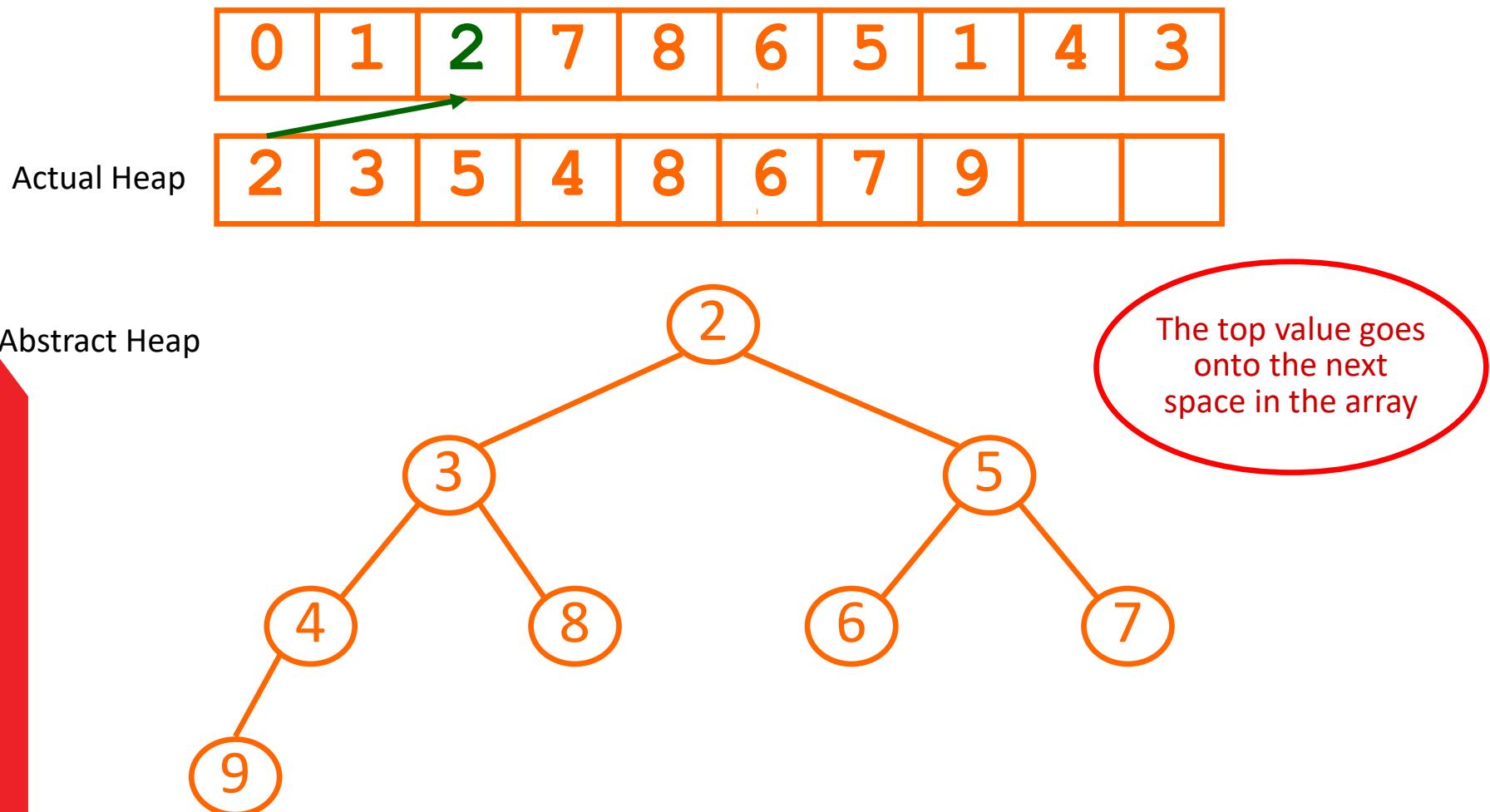
Heap Delete Animation



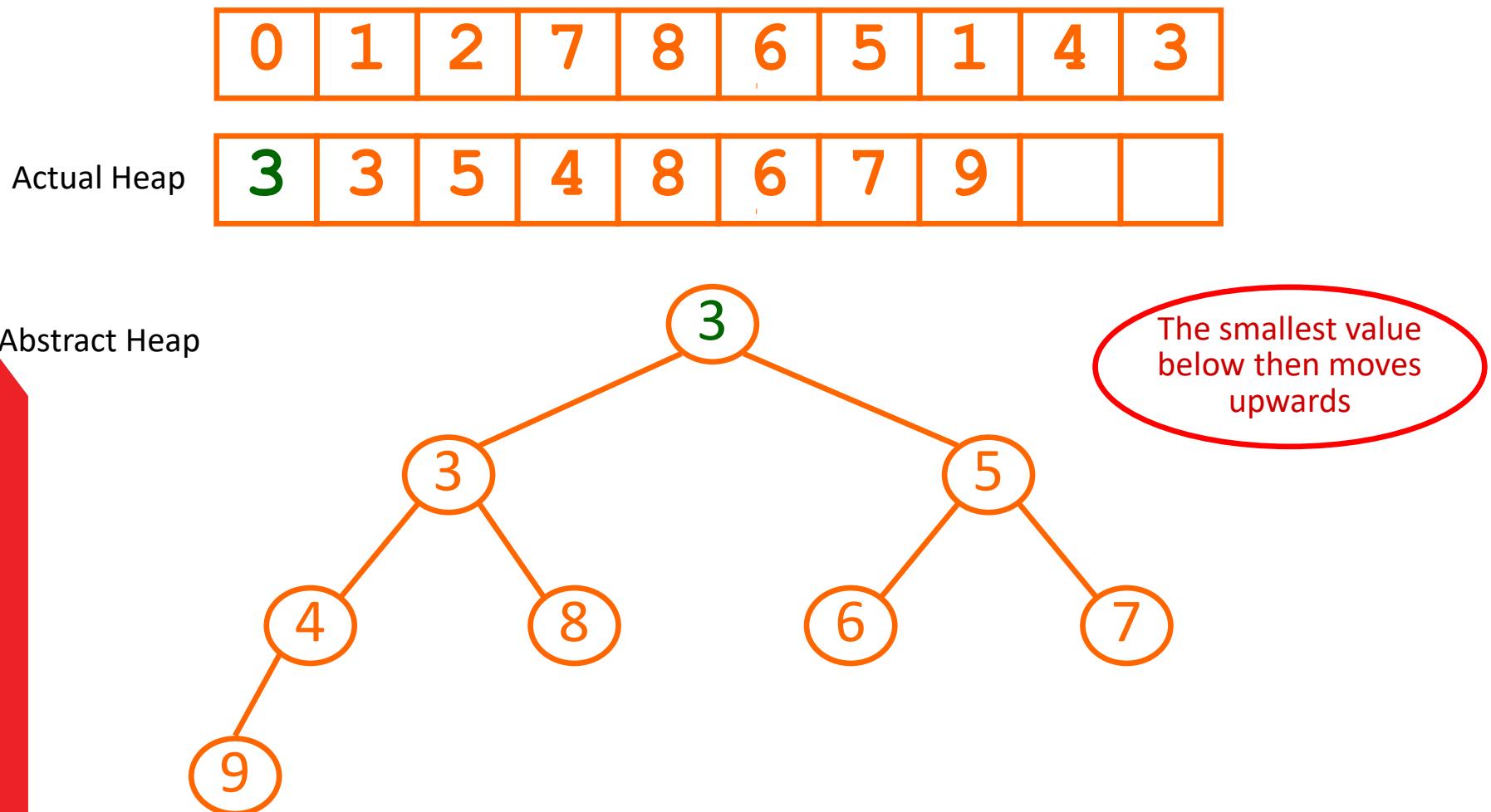
Heap Delete Animation



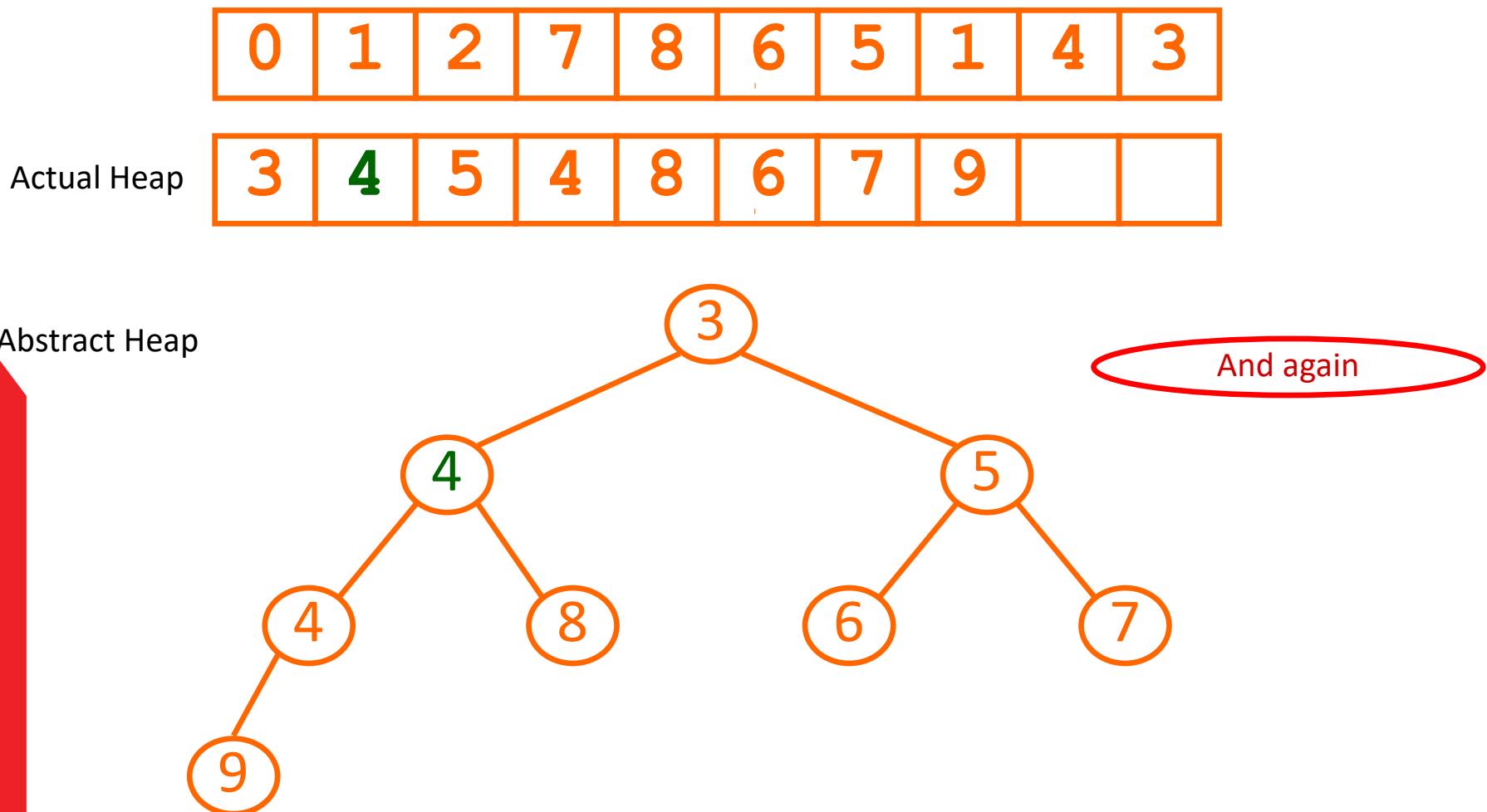
Heap Delete Animation



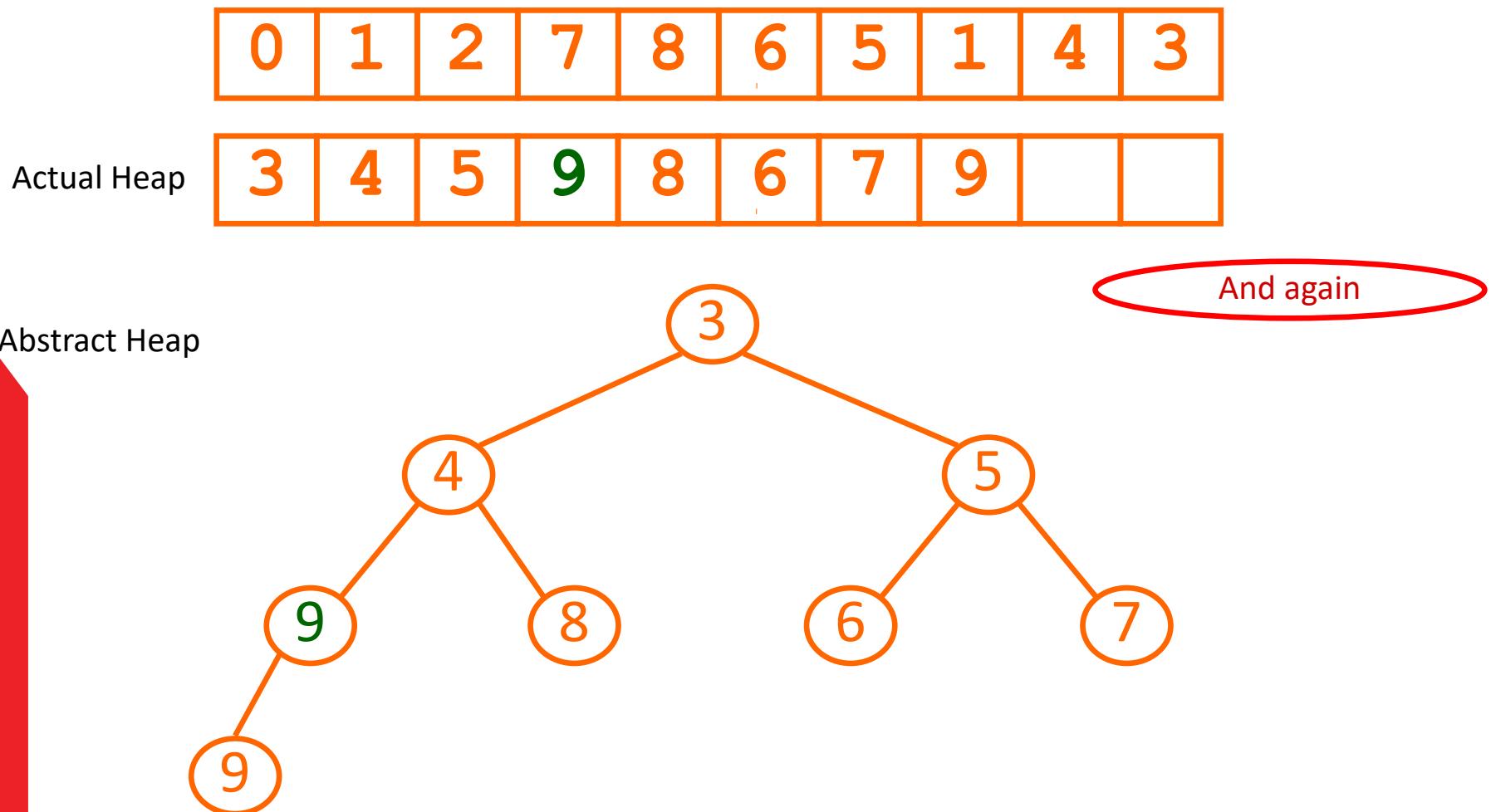
Heap Delete Animation



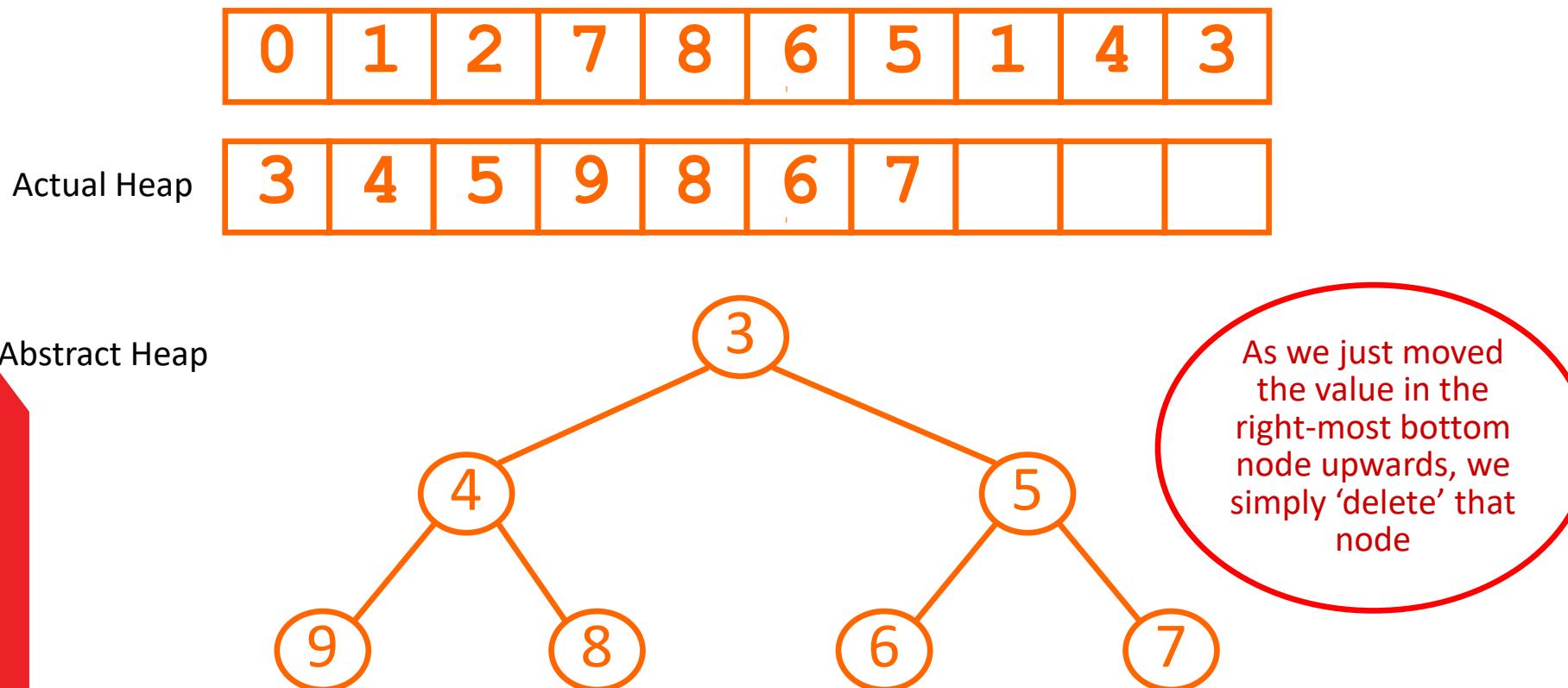
Heap Delete Animation



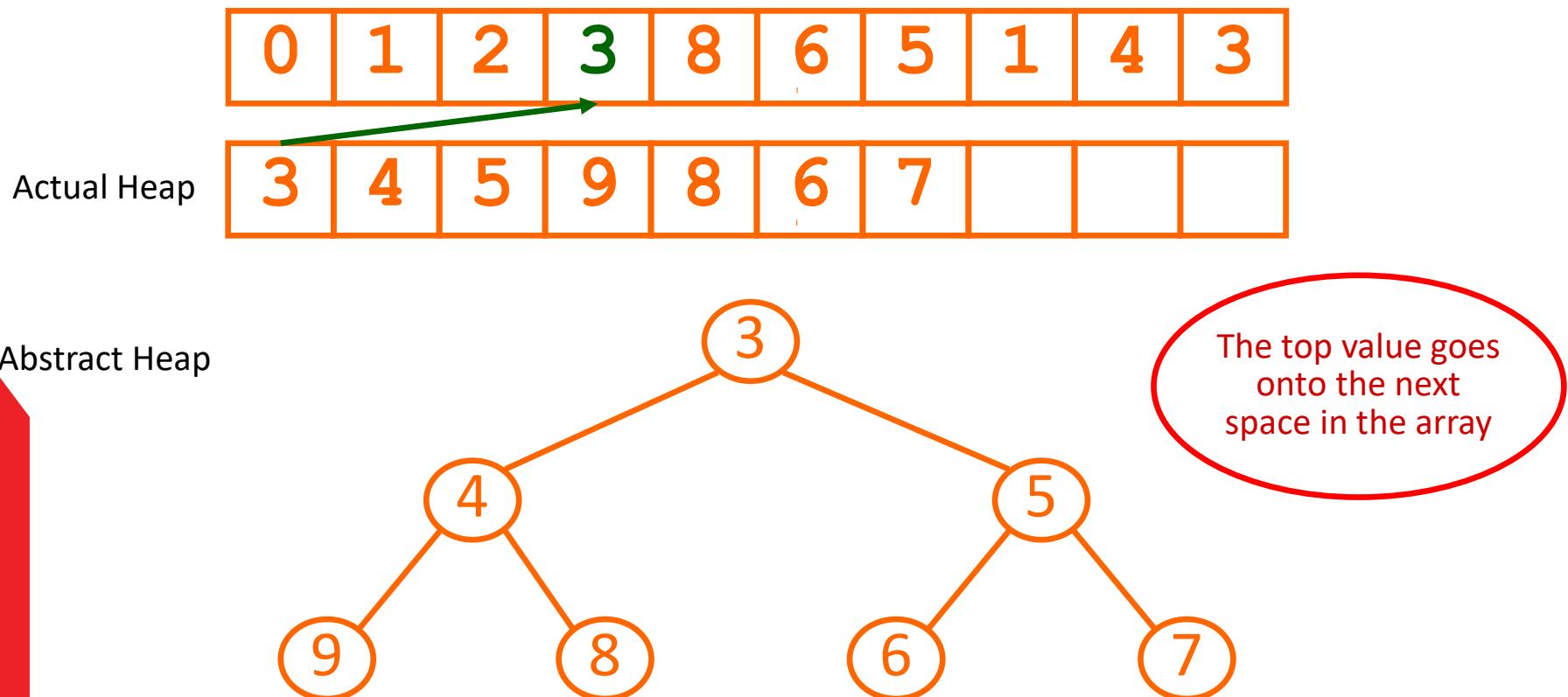
Heap Delete Animation



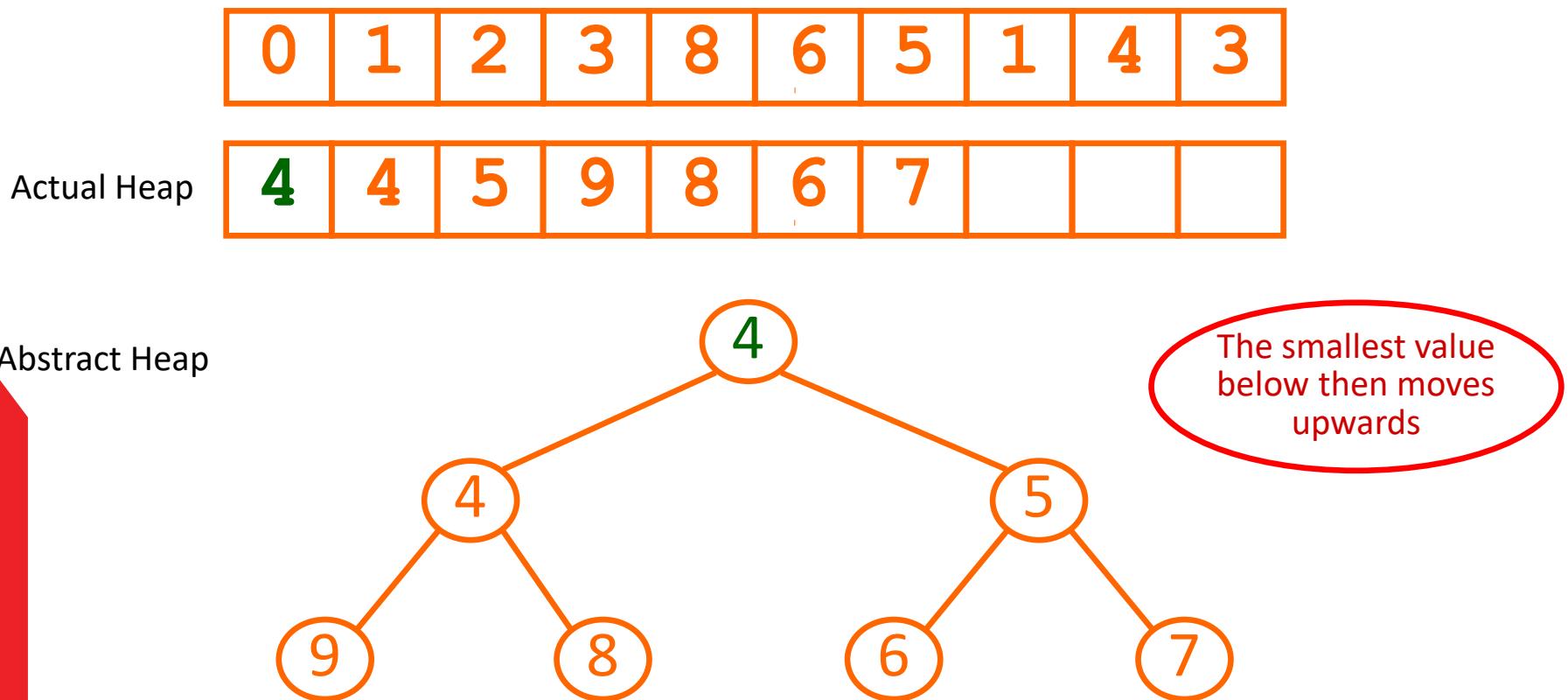
Heap Delete Animation



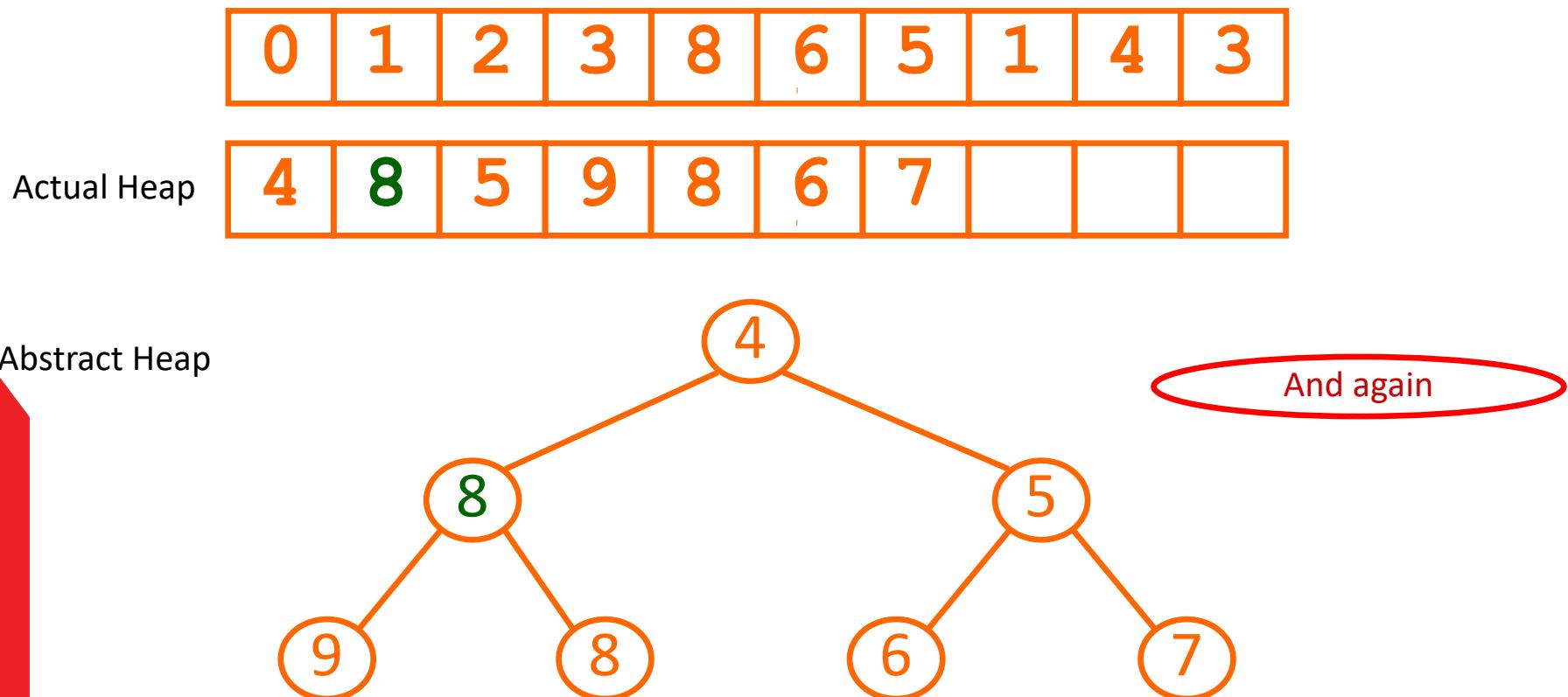
Heap Delete Animation



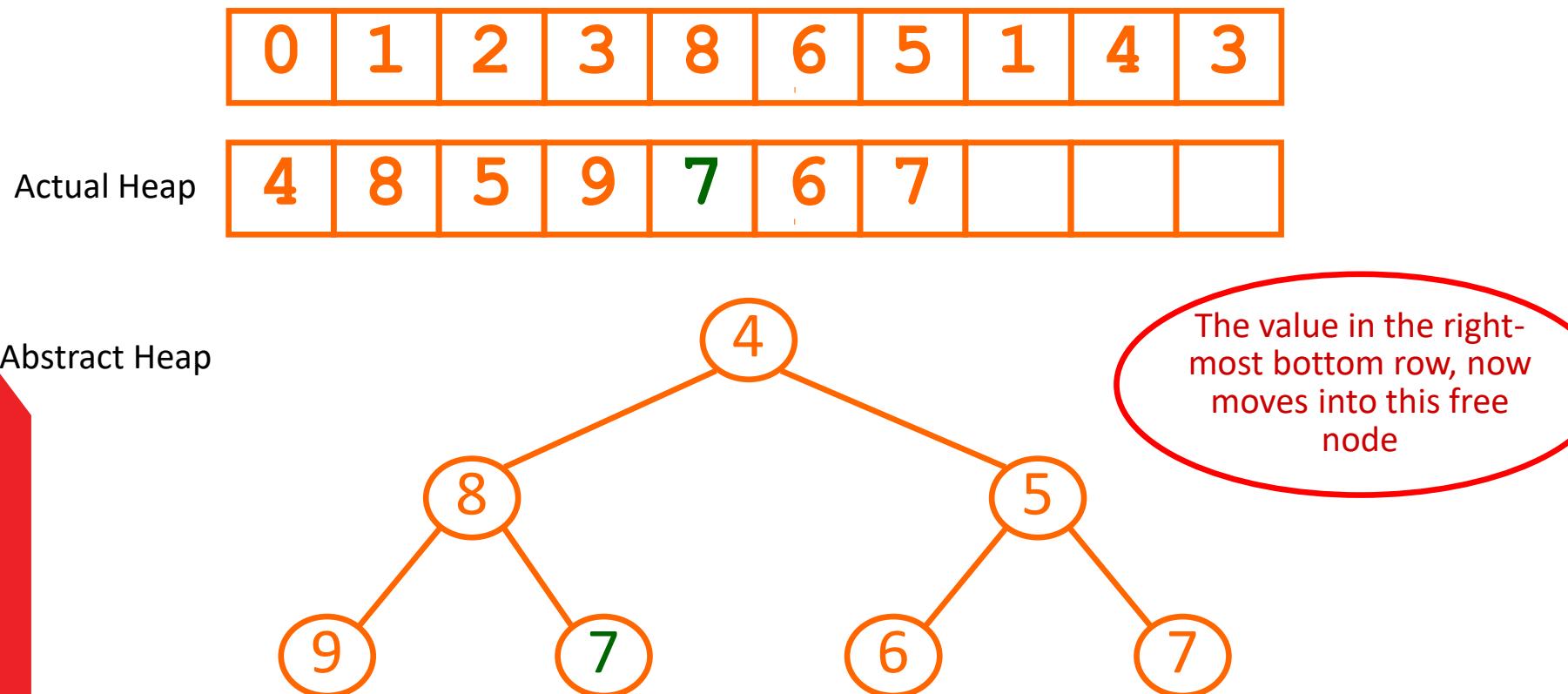
Heap Delete Animation



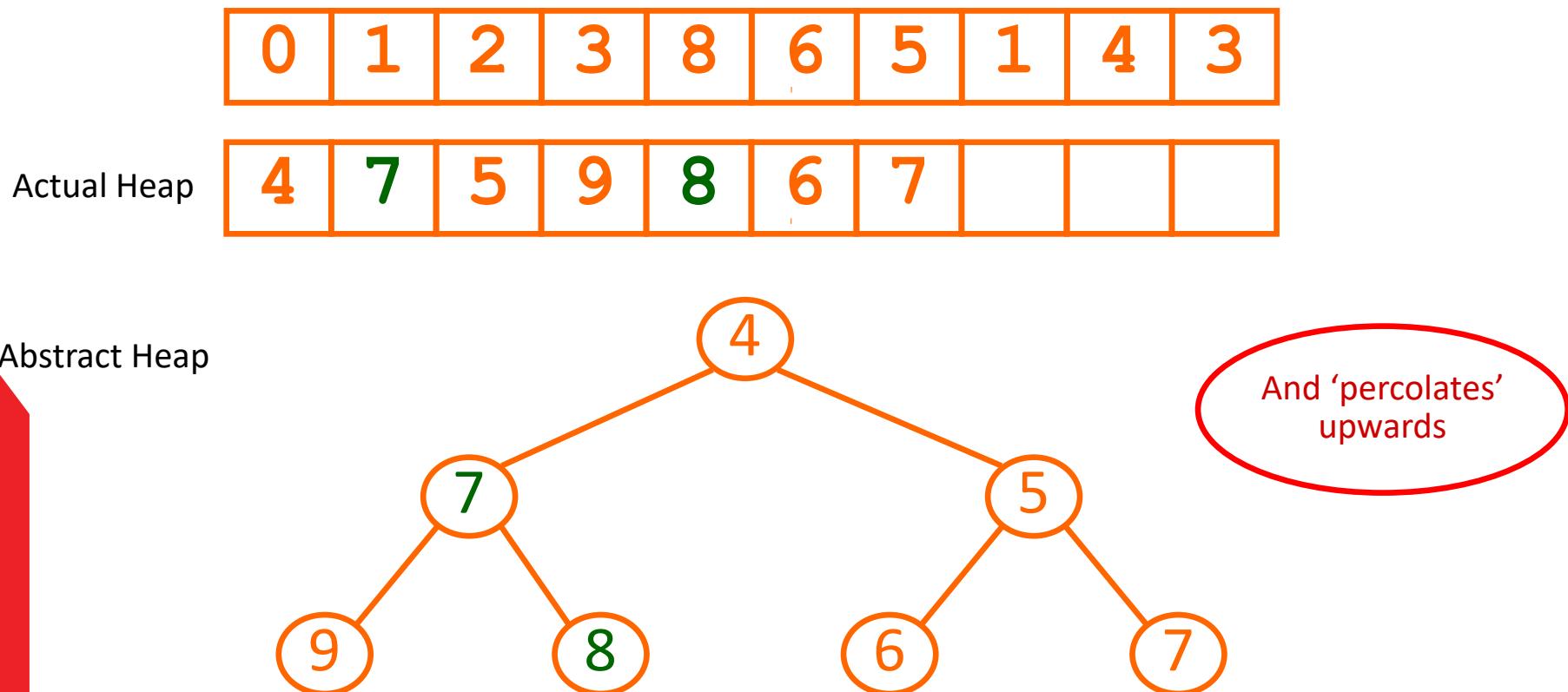
Heap Delete Animation



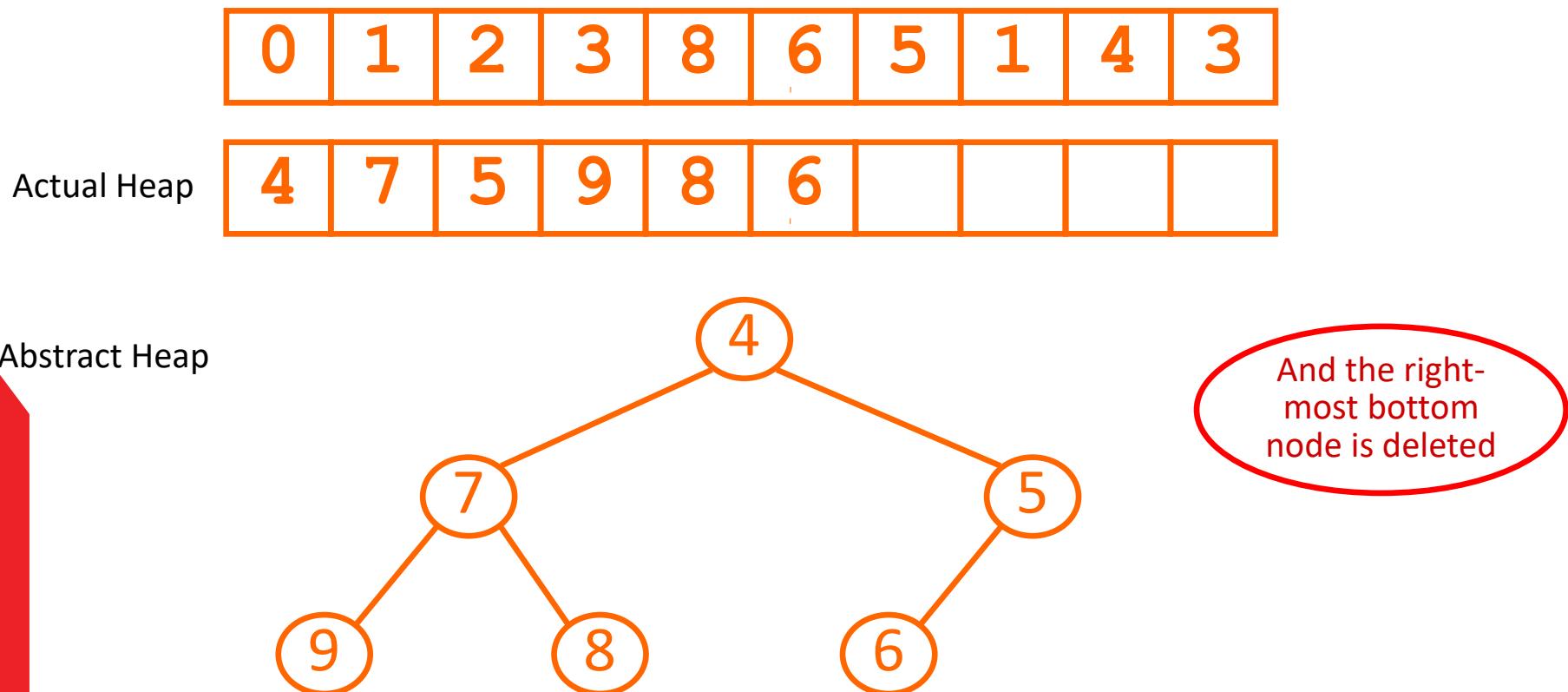
Heap Delete Animation



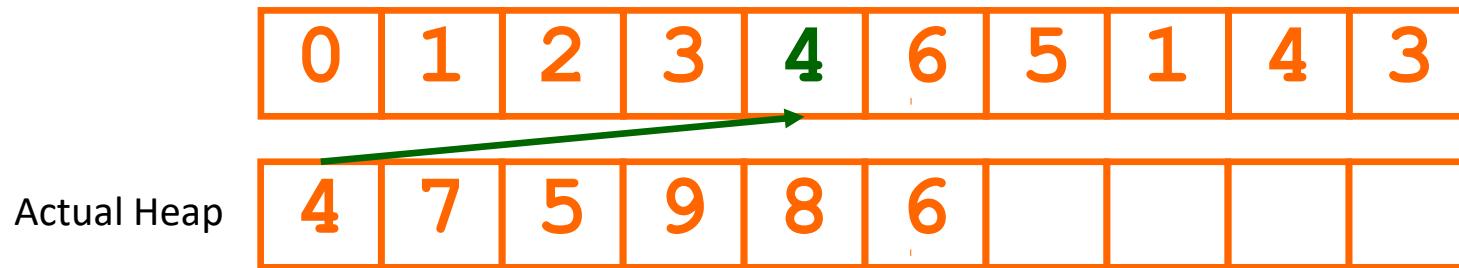
Heap Delete Animation



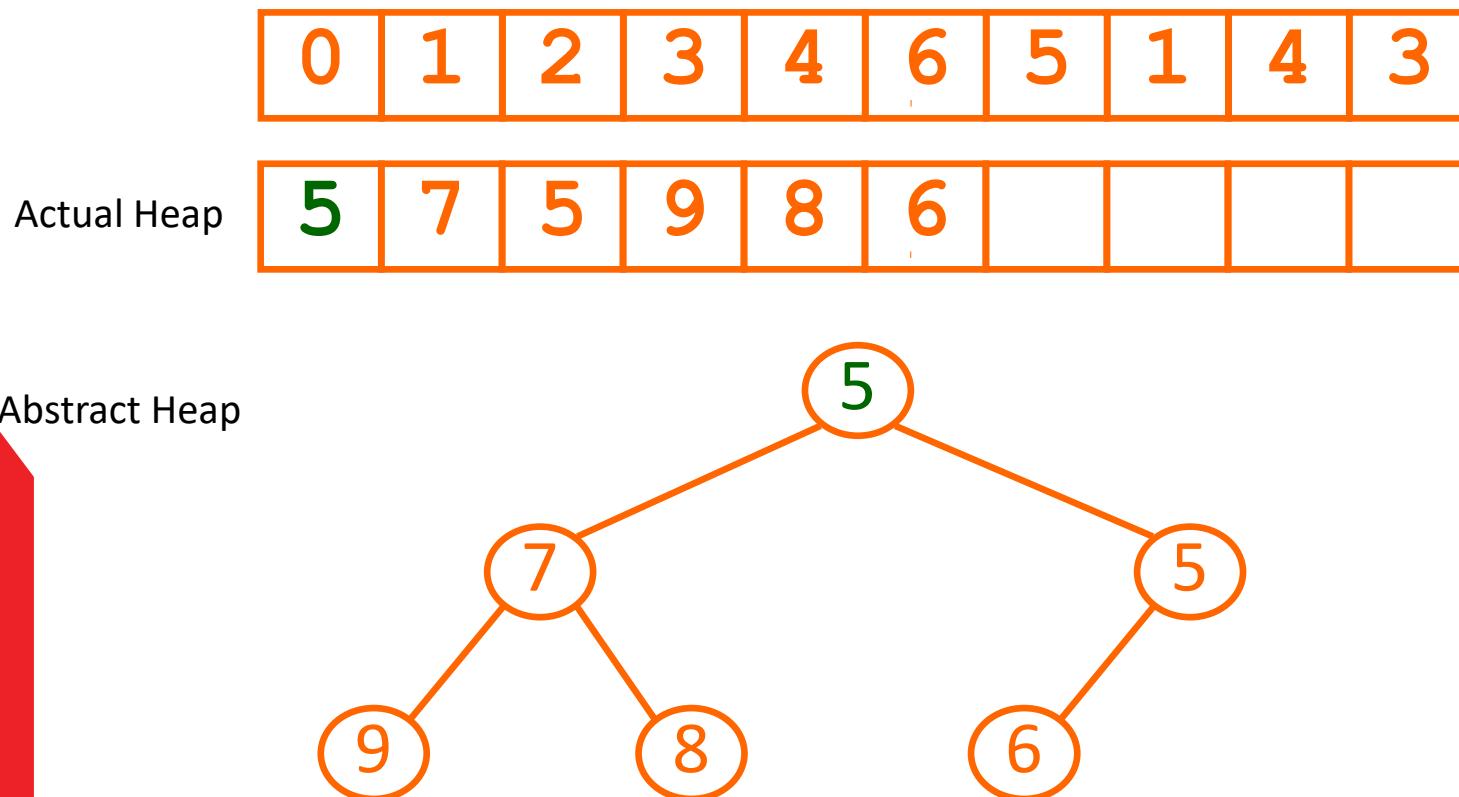
Heap Delete Animation



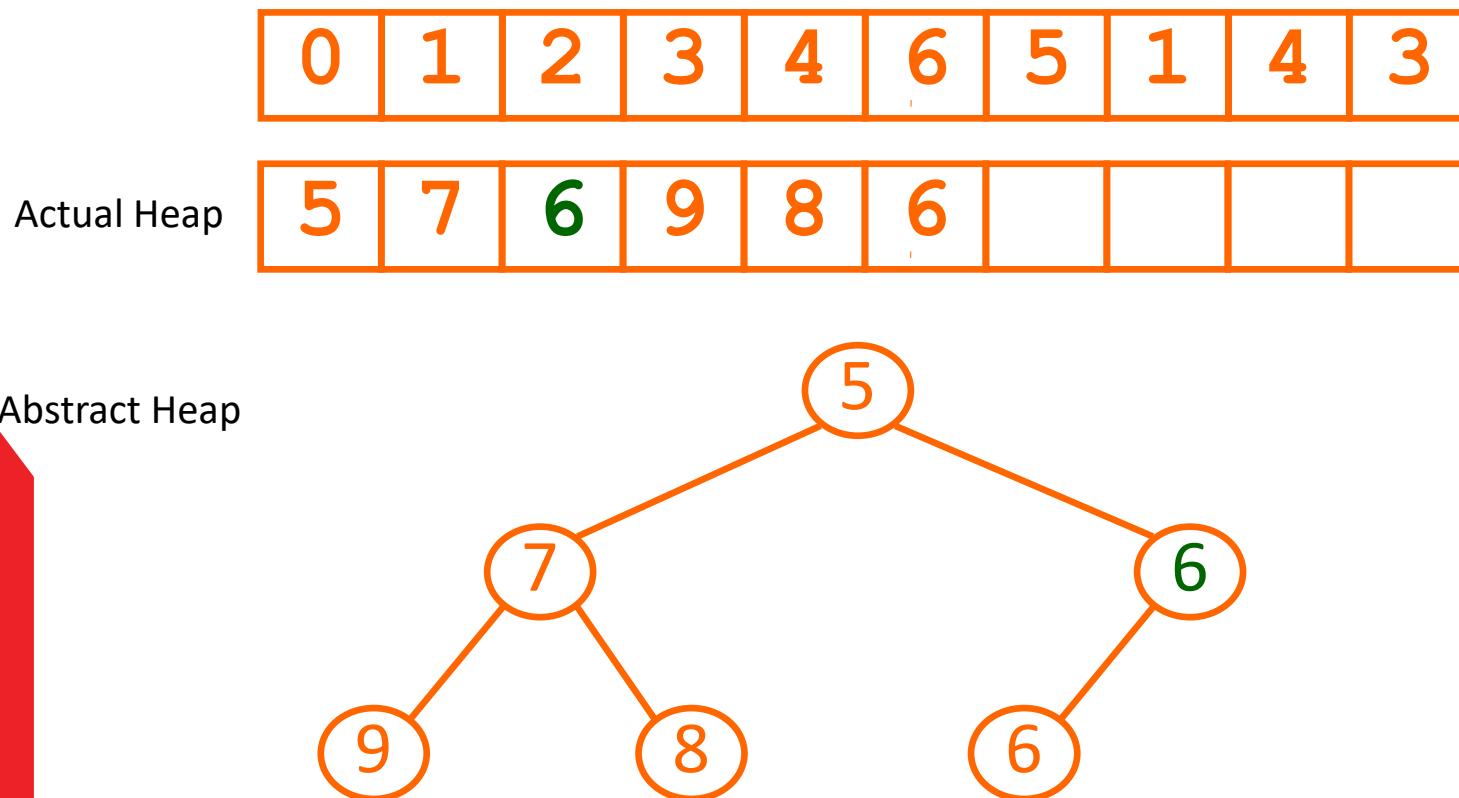
Heap Delete Animation



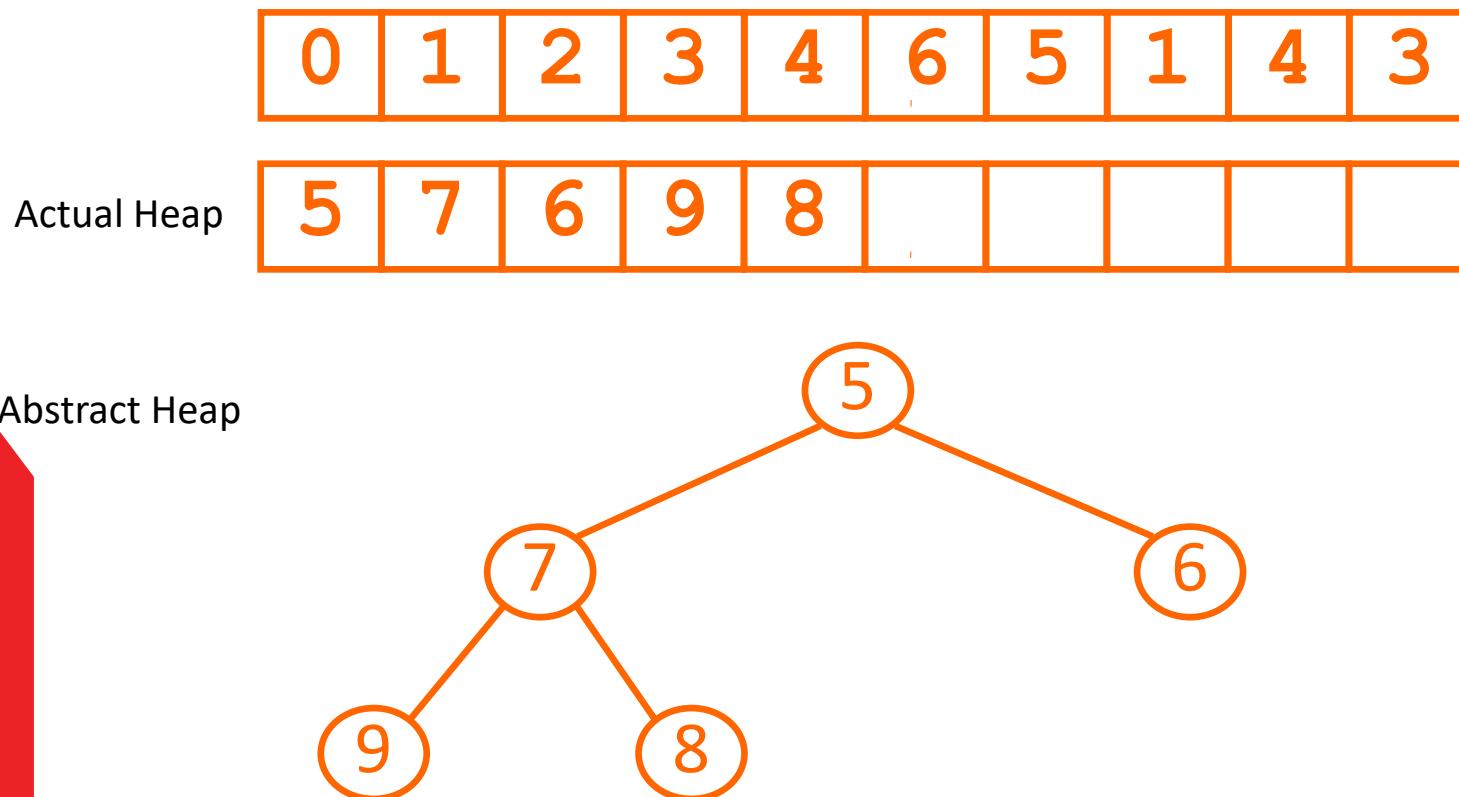
Heap Delete Animation



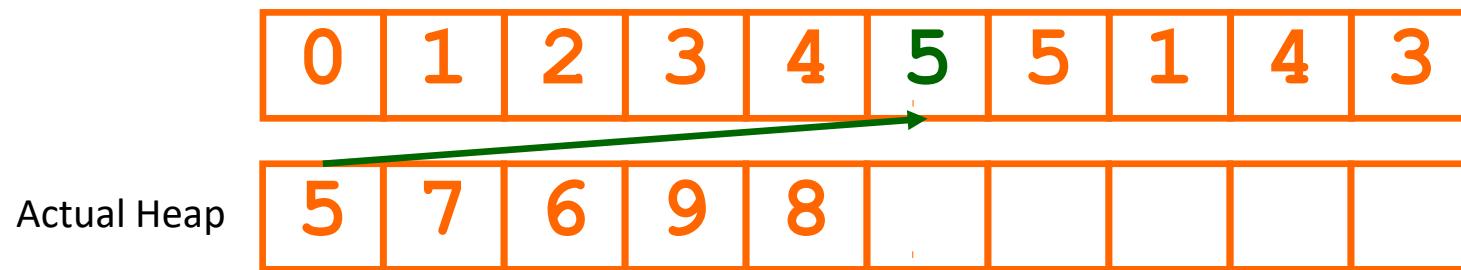
Heap Delete Animation



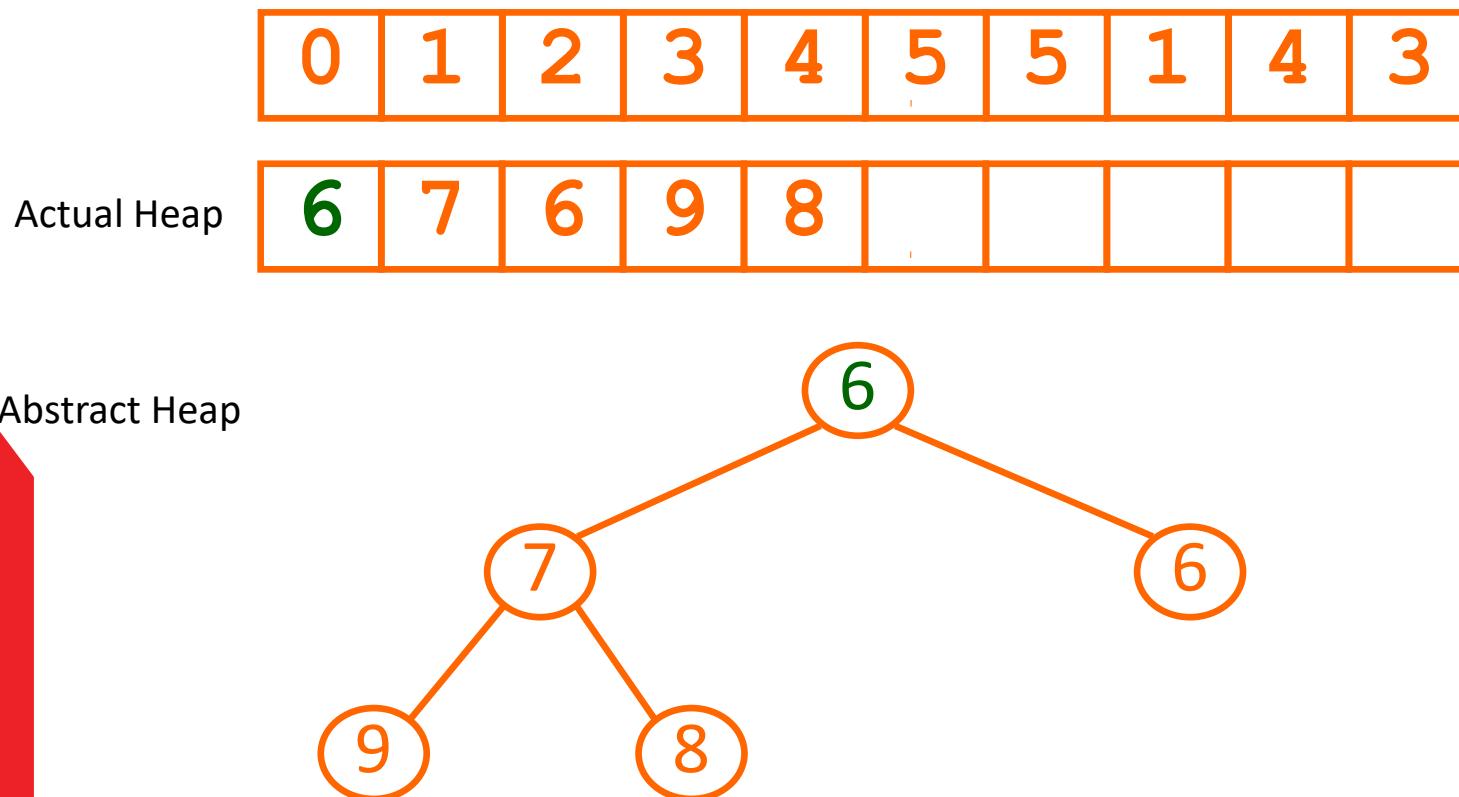
Heap Delete Animation



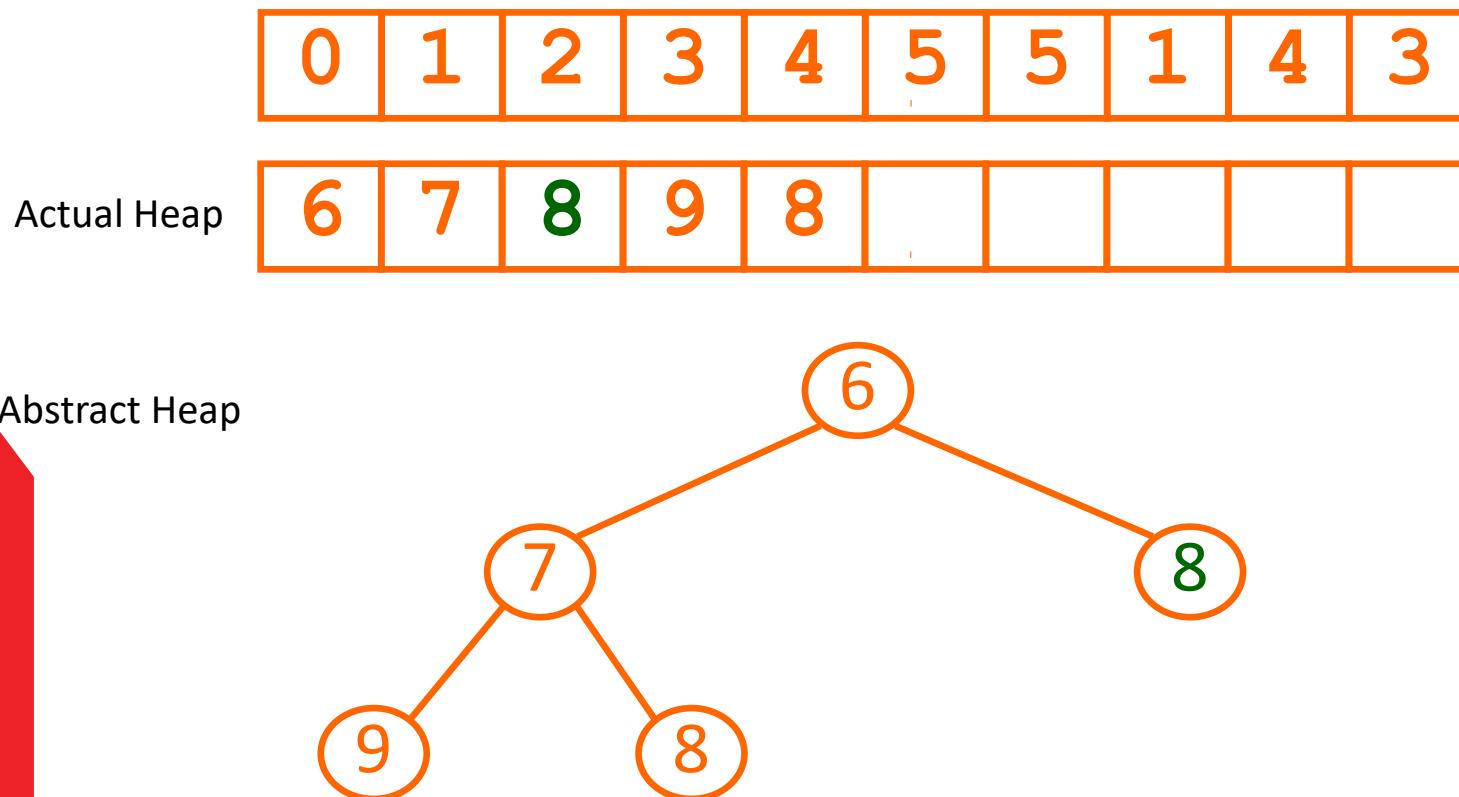
Heap Delete Animation



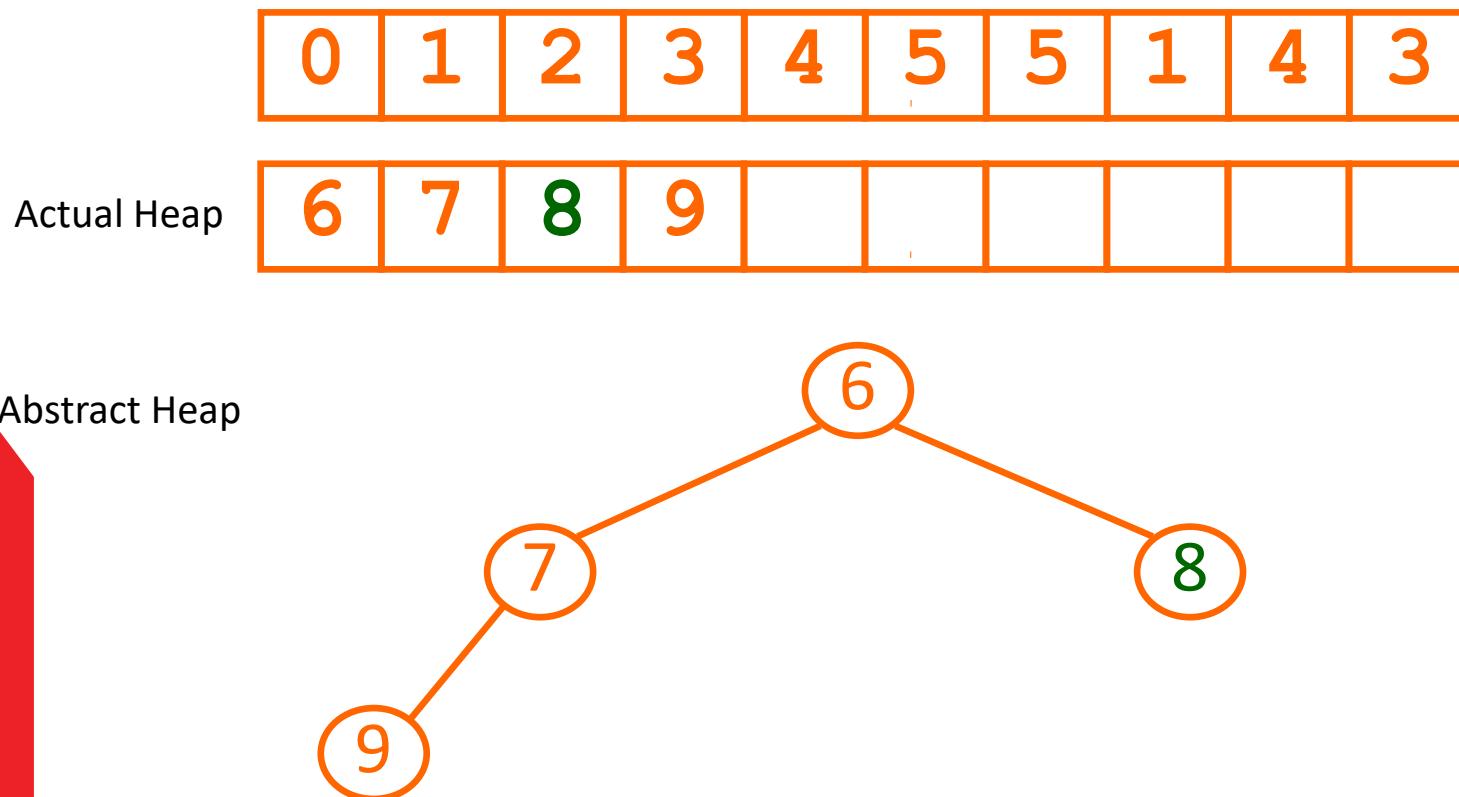
Heap Delete Animation



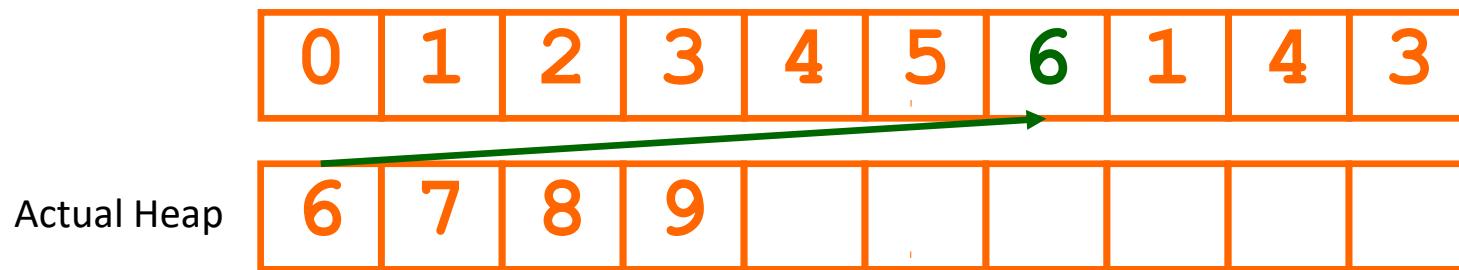
Heap Delete Animation



Heap Delete Animation

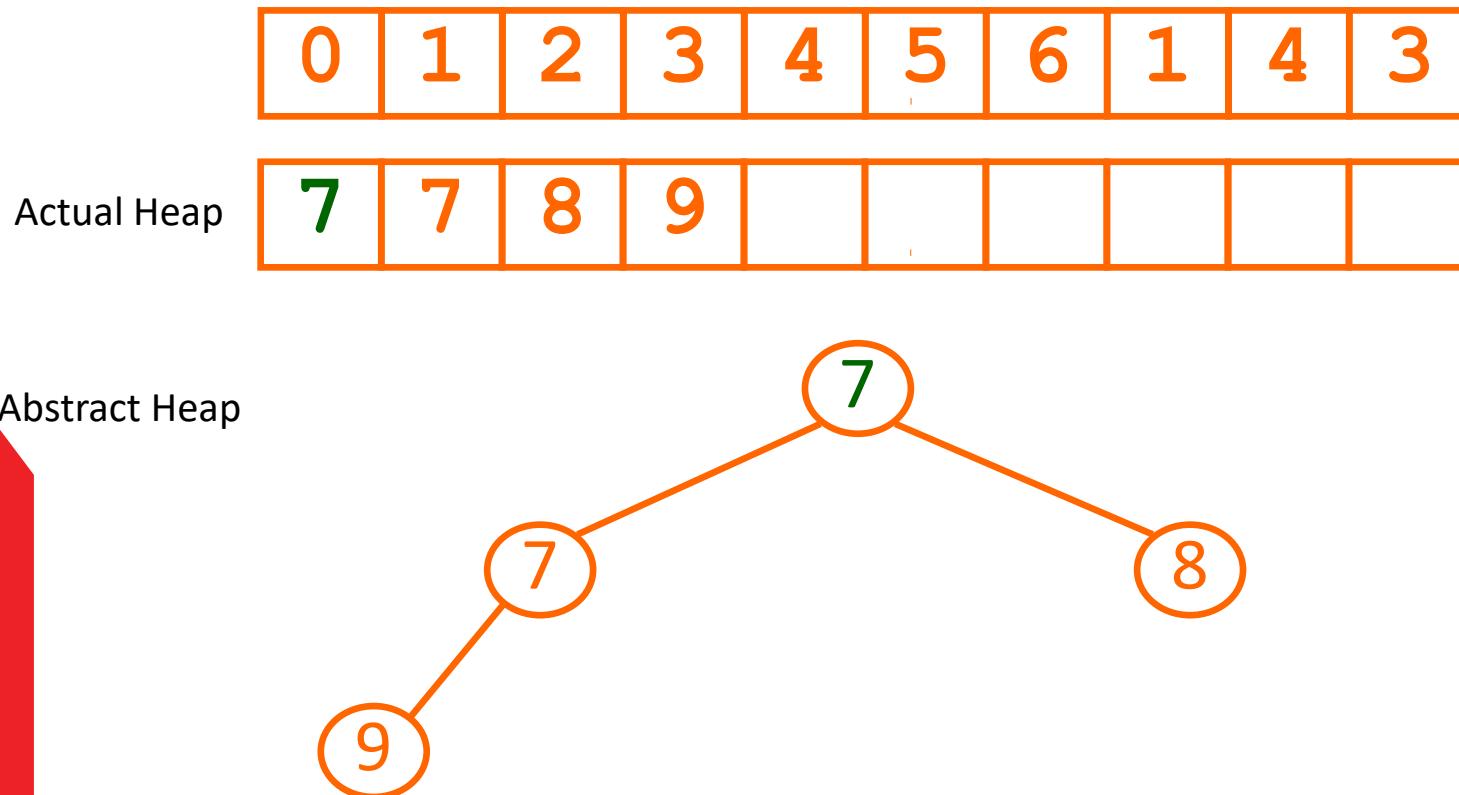


Heap Delete Animation

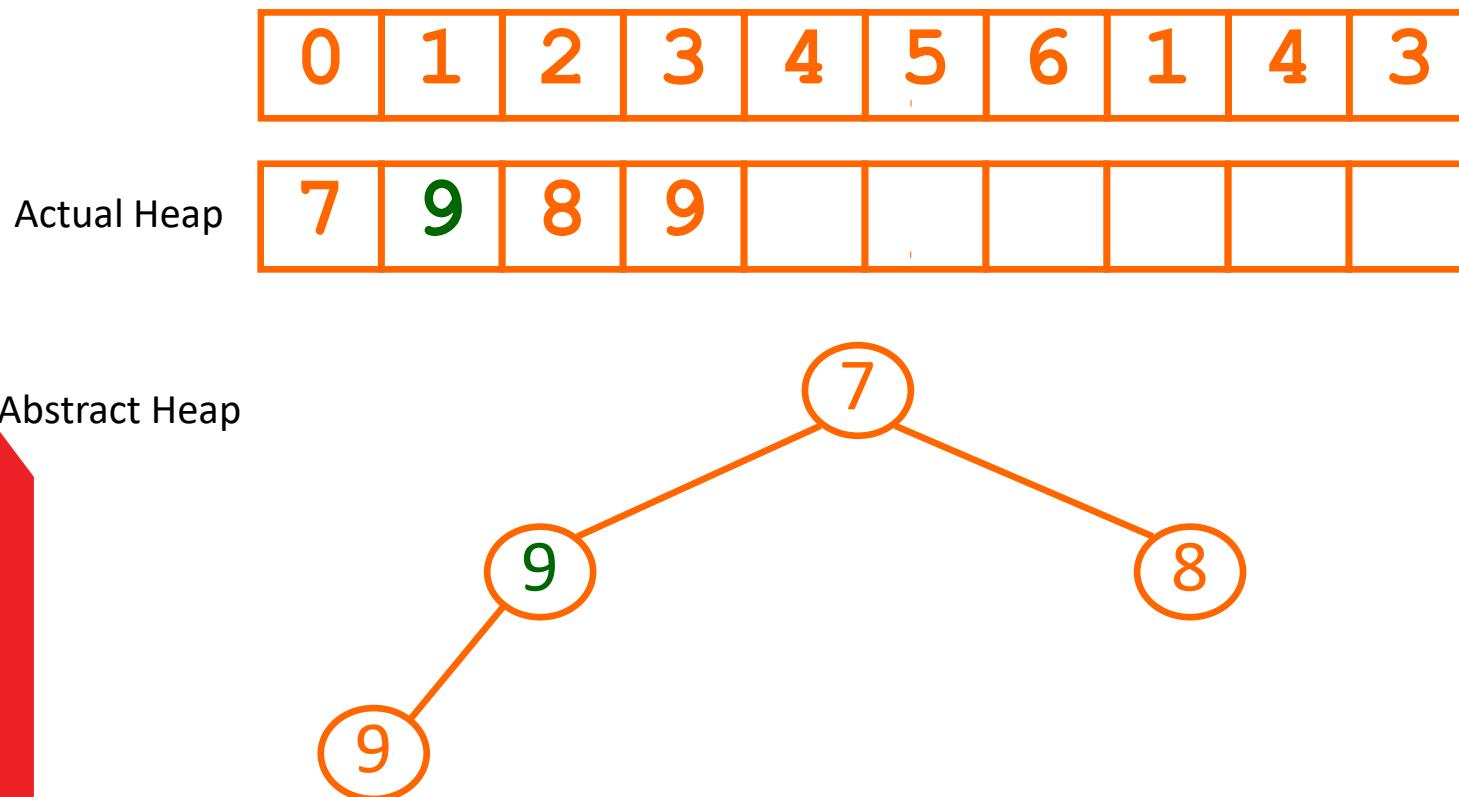


Abstract Heap

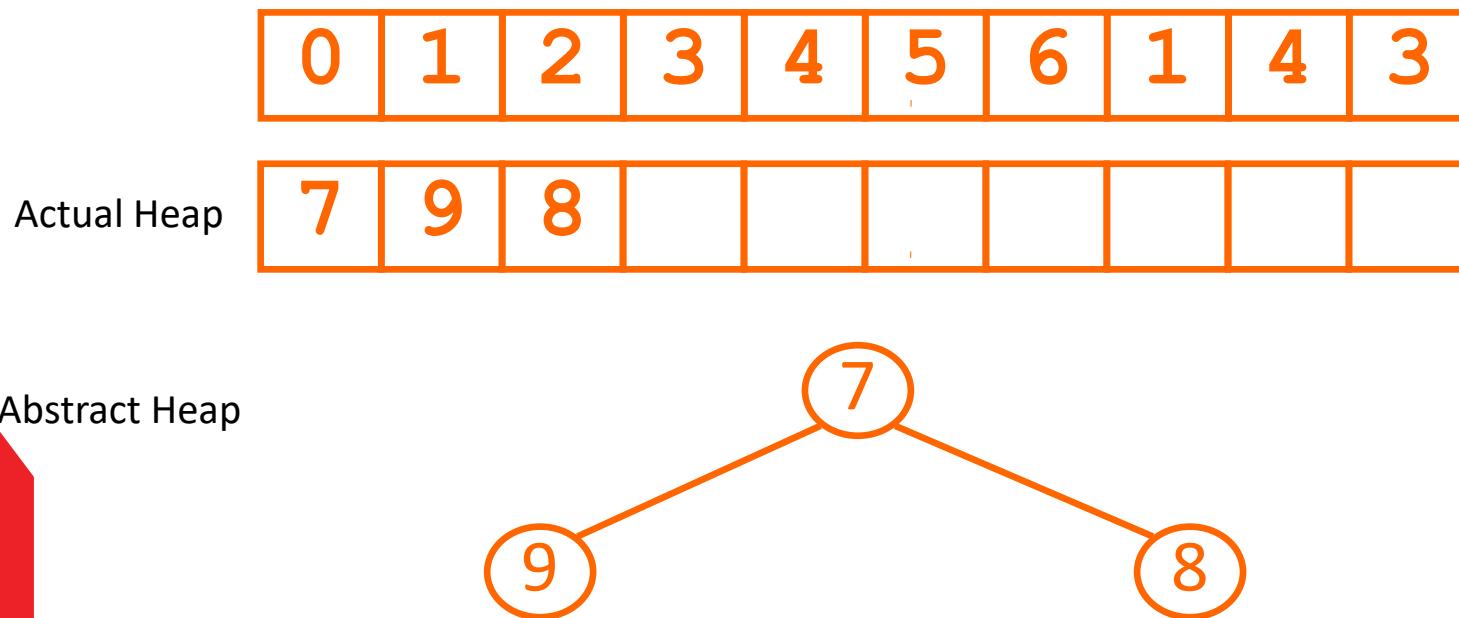
Heap Delete Animation



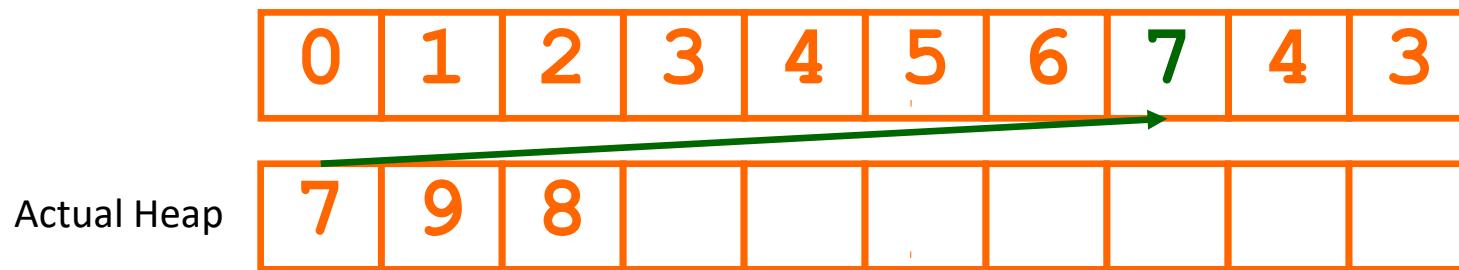
Heap Delete Animation



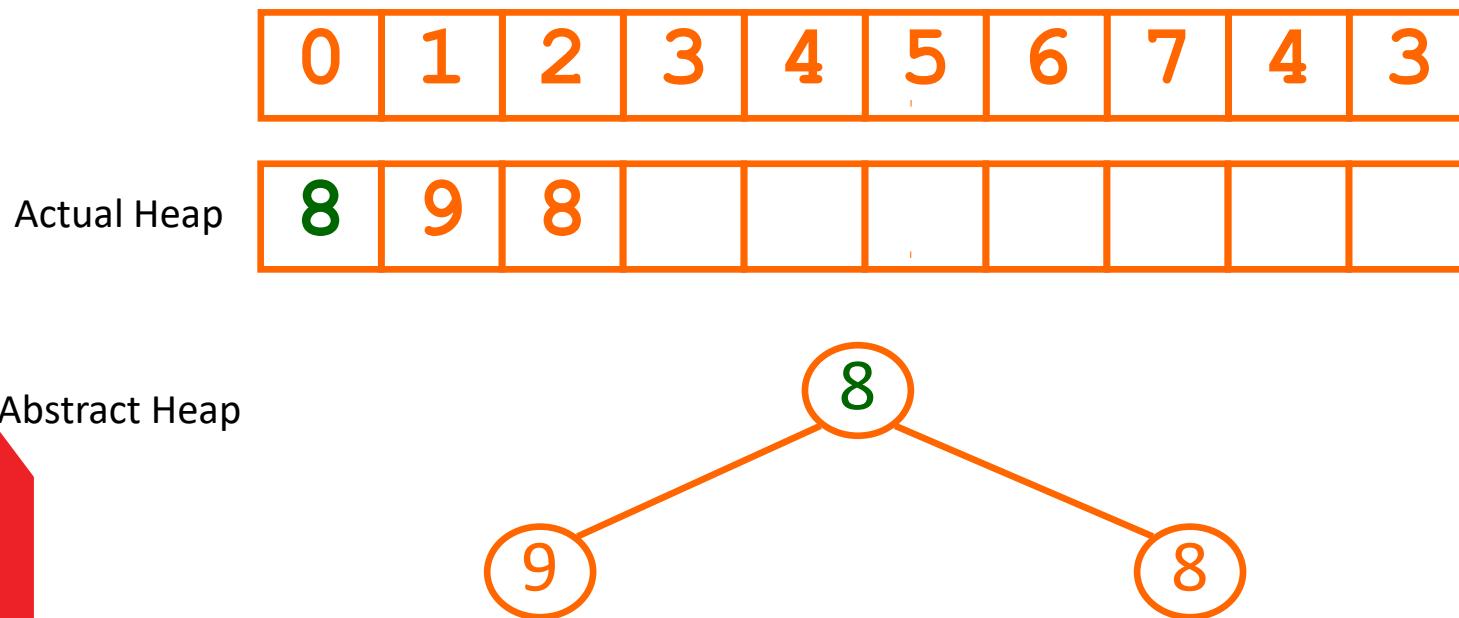
Heap Delete Animation



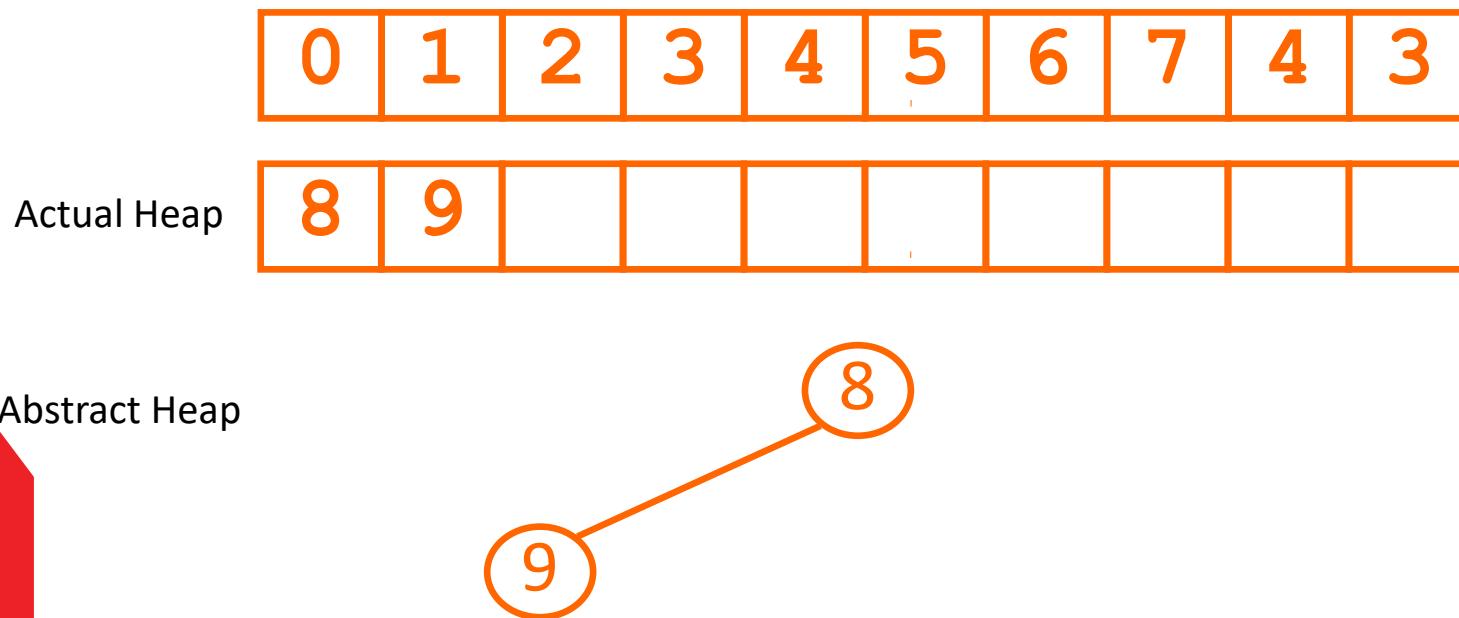
Heap Delete Animation



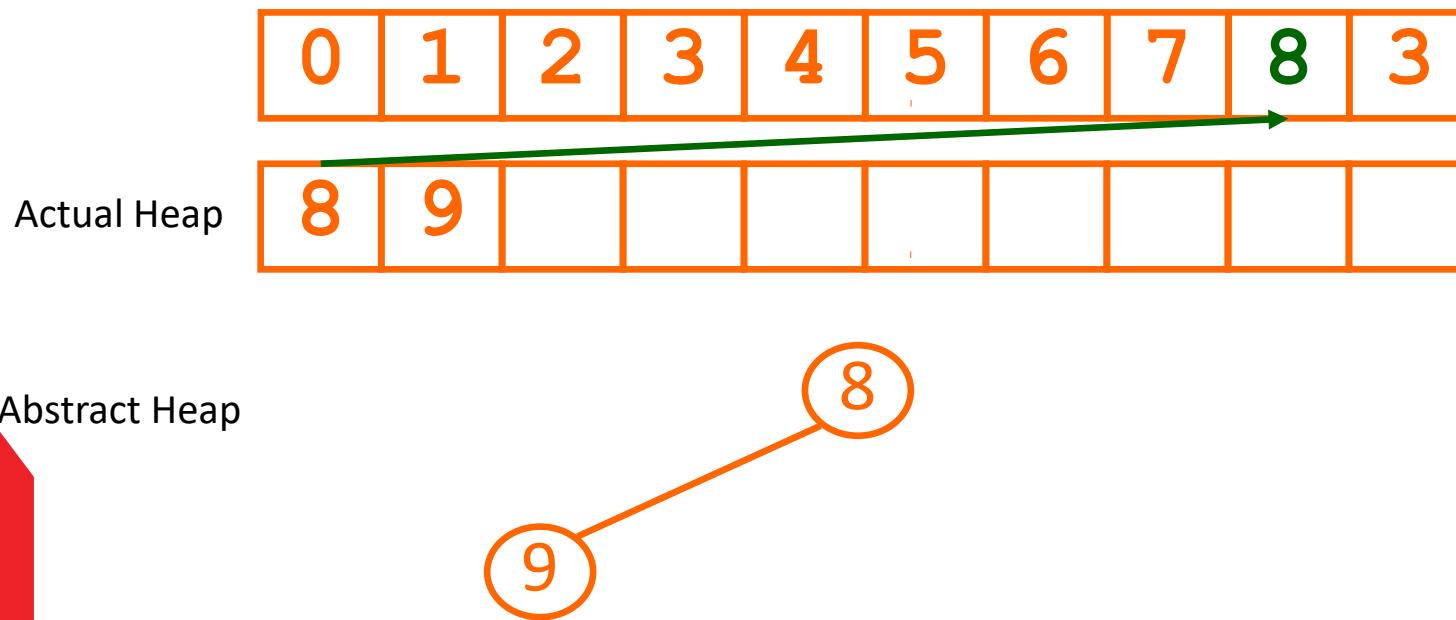
Heap Delete Animation



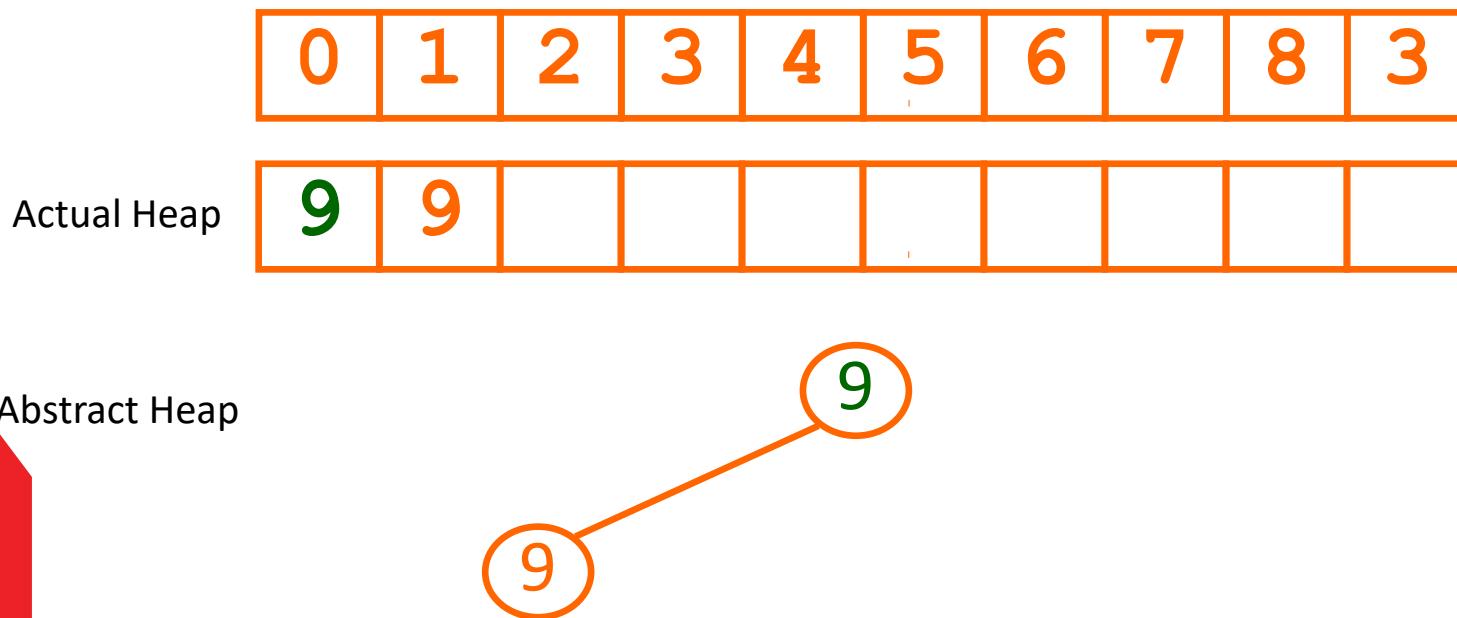
Heap Delete Animation



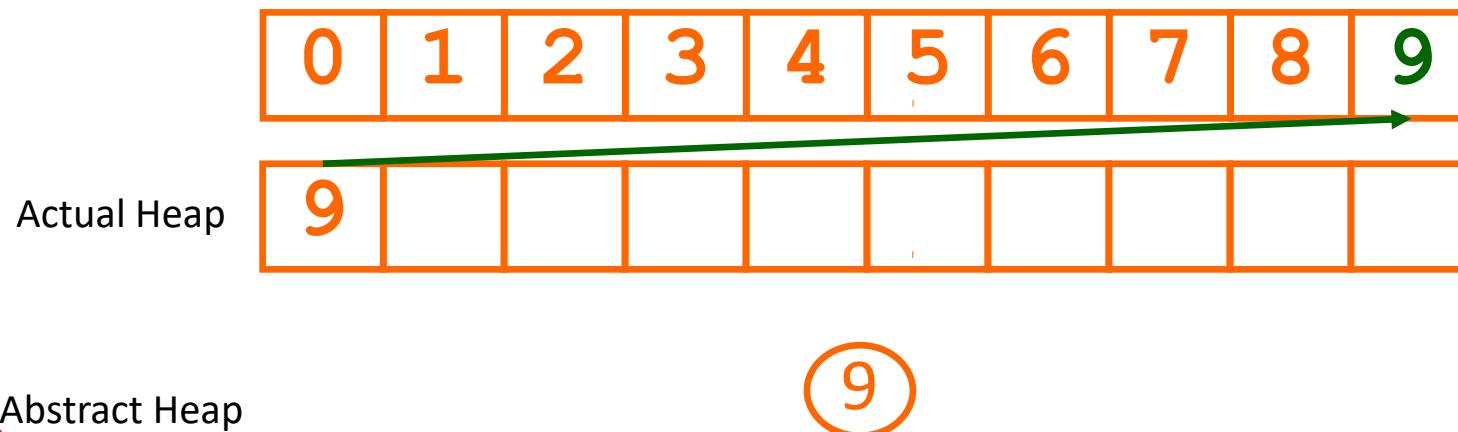
Heap Delete Animation



Heap Delete Animation



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.

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
```

```

• 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

Quicksort Animation

Split()

pivotValue



Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



index1

FOUND

index2

Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



index1

FOUND.

index2

Quicksort Animation

Split()

pivotValue



index1

index2

Swap them

Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



index1

FOUND.

index2

Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



index1

index2

index2 reaches
index1, so we halt

Quicksort Animation

Split()

pivotIndex

Set
pivotIndex to
index2-1

pivotValue



index1

index2

Quicksort Animation

Split()

pivotIndex

Swap the
pivotValue
and the value
at the
pivotIndex

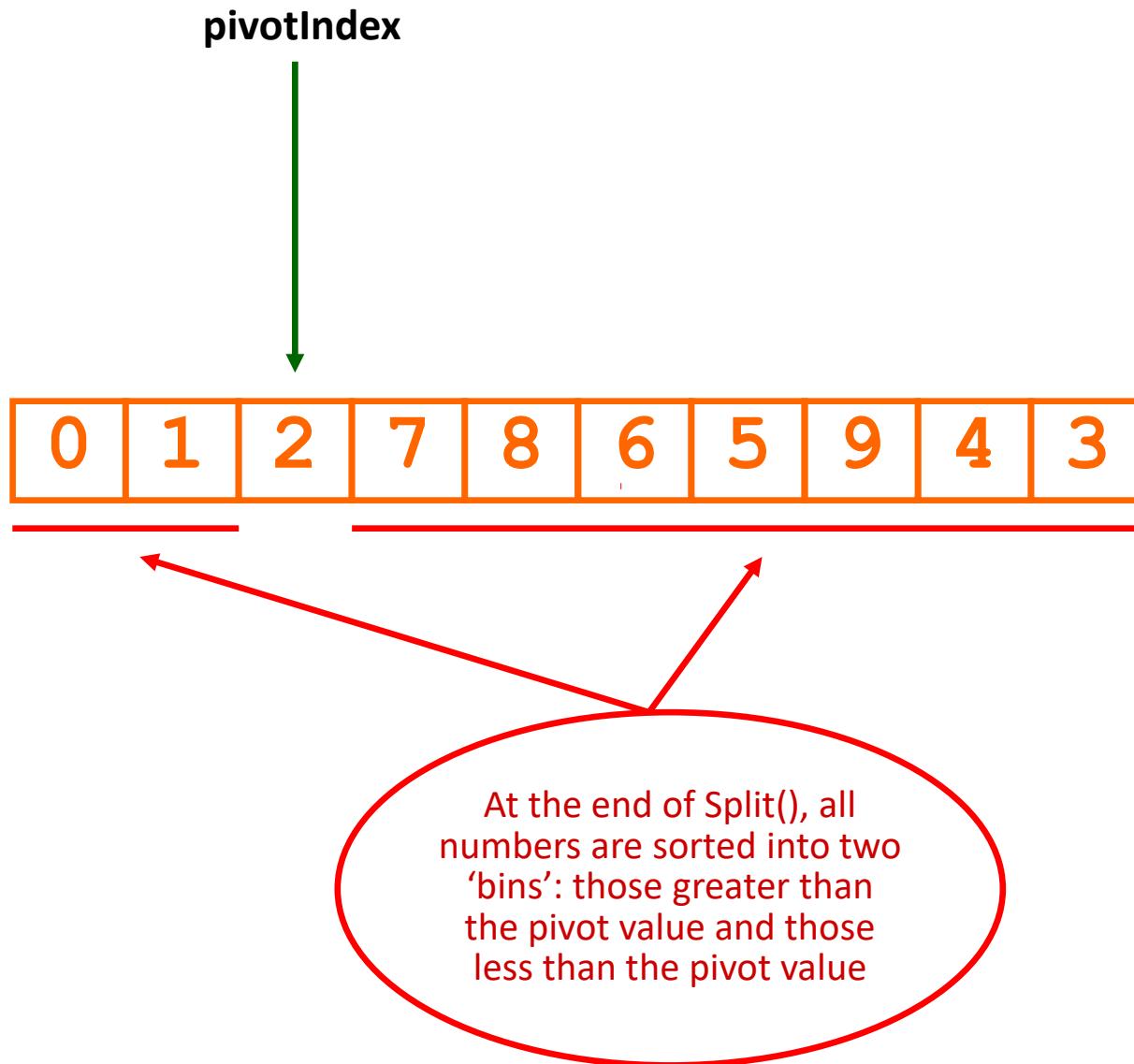
pivotValue



index1

index2

Quicksort Animation



Quicksort Animation

pivotIndex



Quicksort the
section below
pivotIndex

Quicksort Animation

pivotIndex



Less than three
elements and
already in order

Quicksort Animation



Quicksort the
section above
pivotIndex

Quicksort Animation

Split()

pivotValue



Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



FOUND

index1

index2

Quicksort Animation

Split()

pivotValue



Quicksort Animation

Split()

pivotValue



index1

FOUND

index2

Quicksort Animation

Split()

pivotValue



index1

Swap them

index2

Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



Quicksort Animation

Split()

pivotValue



index1
index2

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

Quicksort Animation

Split()

pivotValue



index1

index2

FOUND

Quicksort Animation

Split()

pivotValue



Swap them

index1

index2

Quicksort Animation

Split()

pivotValue



index1
reaches
index2 so
we halt

index1
index2

Quicksort Animation

Split()

Set
pivotIndex
to
index2-1

pivotValue



index1

index2

Quicksort Animation

Split()

pivotValue



Swap the
pivotValue and
the value at the
pivotIndex

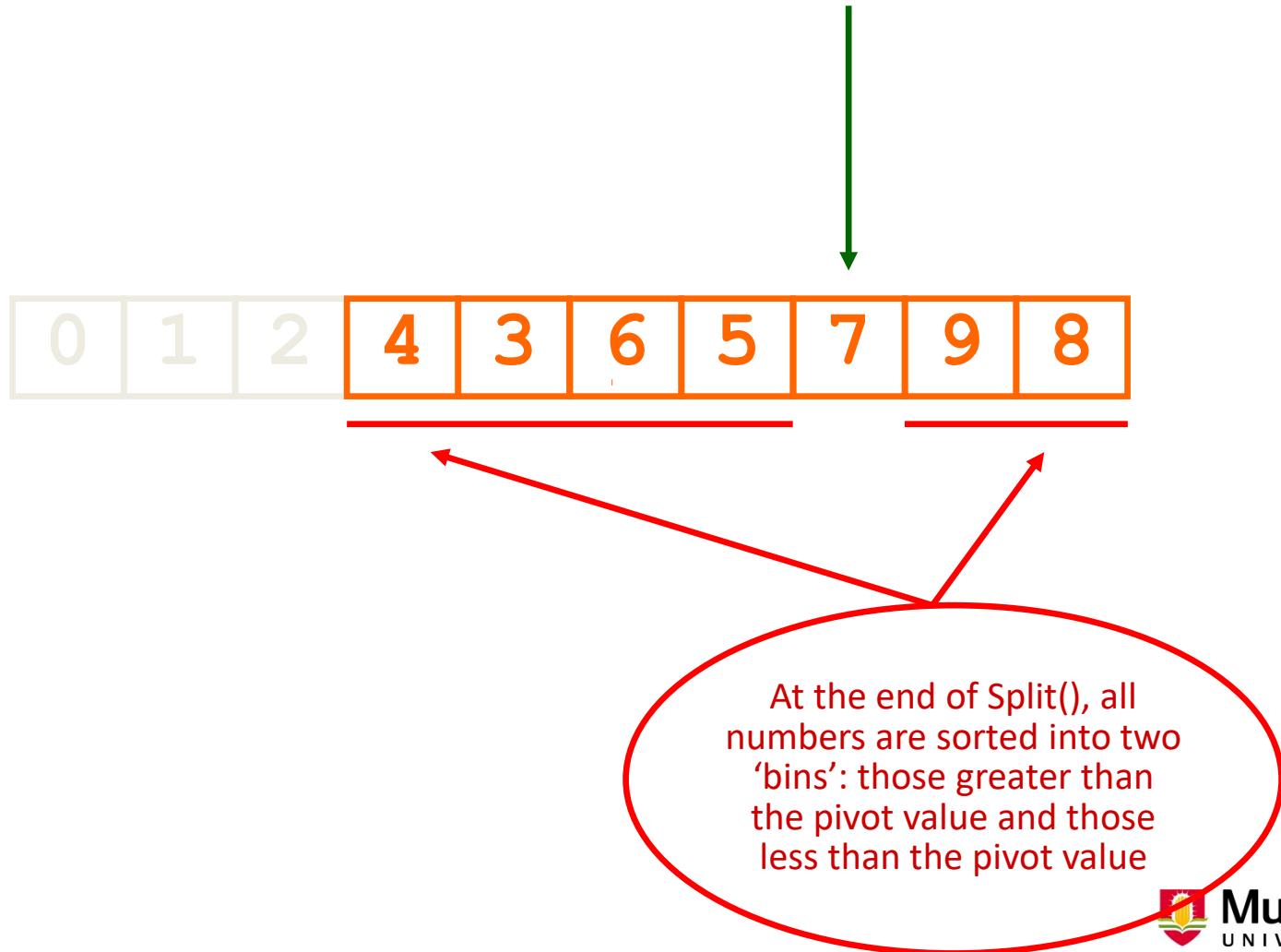
pivotIndex

index1

index2

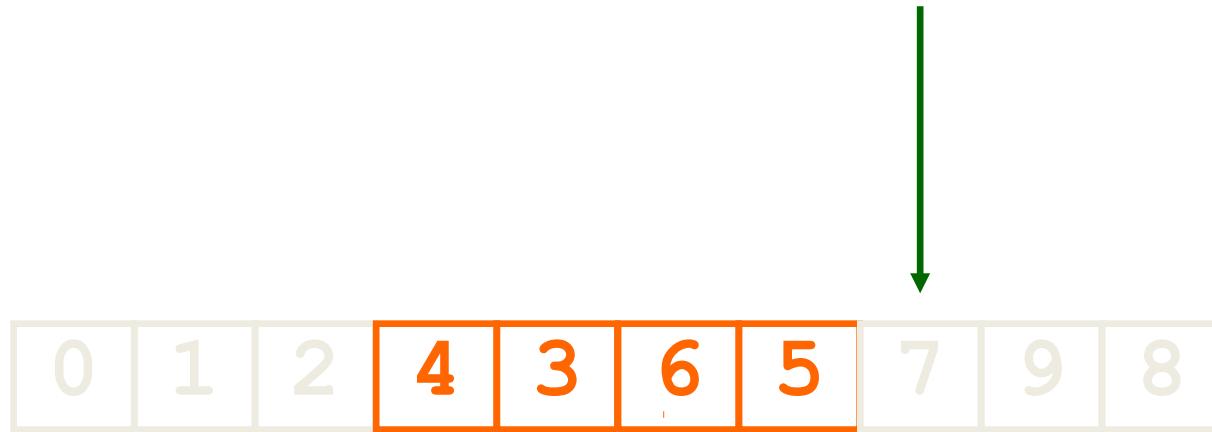
Quicksort Animation

pivotIndex (2)



Quicksort Animation

pivotIndex (2)



Quicksort the
section below the
pivotIndex

Quicksort Animation

Split()

pivotValue



Quicksort Animation

Split()

pivotValue



index1

index2

Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



FOUND

index1

index2

Quicksort Animation

Split()

pivotValue



index1

index2

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

Quicksort Animation

Split()

pivotValue



index1
index2

index2 reaches
index1, so we
halt

Quicksort Animation

Split()

pivotValue

pivotIndex

Set pivotIndex
to index2-1



index1
index2

Quicksort Animation

Split()

pivotValue

pivotIndex

Swap the
pivotValue and
the value at
pivotIndex

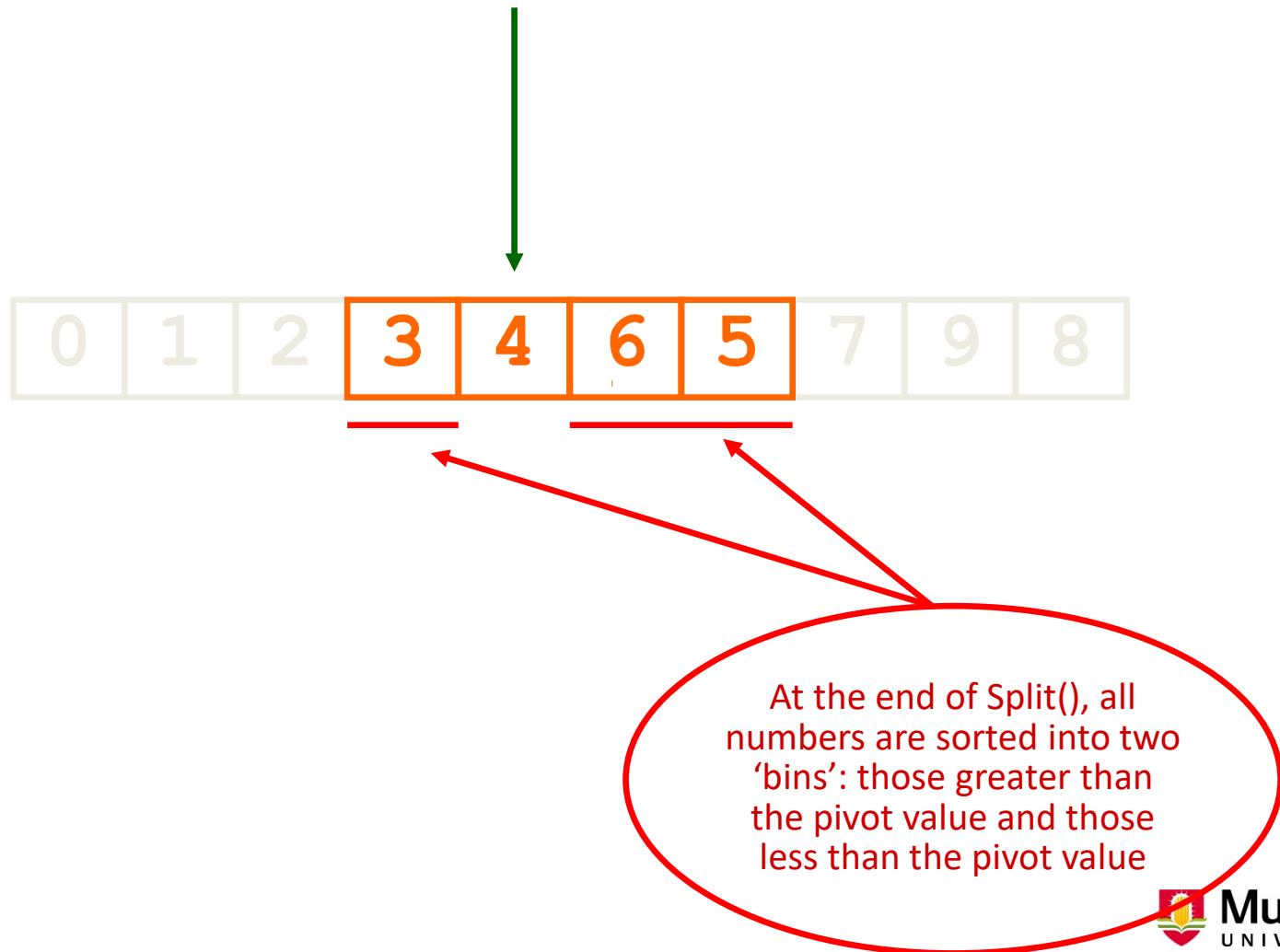


index1

index2

Quicksort Animation

pivotIndex (3)



Quicksort Animation

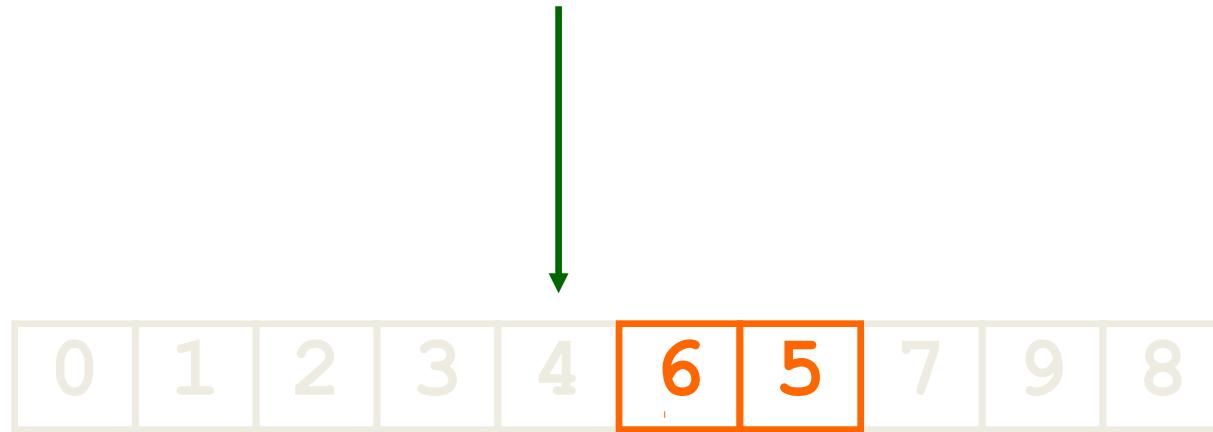
pivotIndex (3)



Only 1 value
below pivotIndex,
so do nothing

Quicksort Animation

pivotIndex (3)



Only two values
above pivotIndex,
but out of order

Quicksort Animation

pivotIndex (3)



So swap them

Quicksort Animation

pivotIndex (2)



Only two values
above pivotIndex,
but out of order

Quicksort Animation

pivotIndex (2)



So swap them

Quicksort Animation

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

Phoenix!!

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

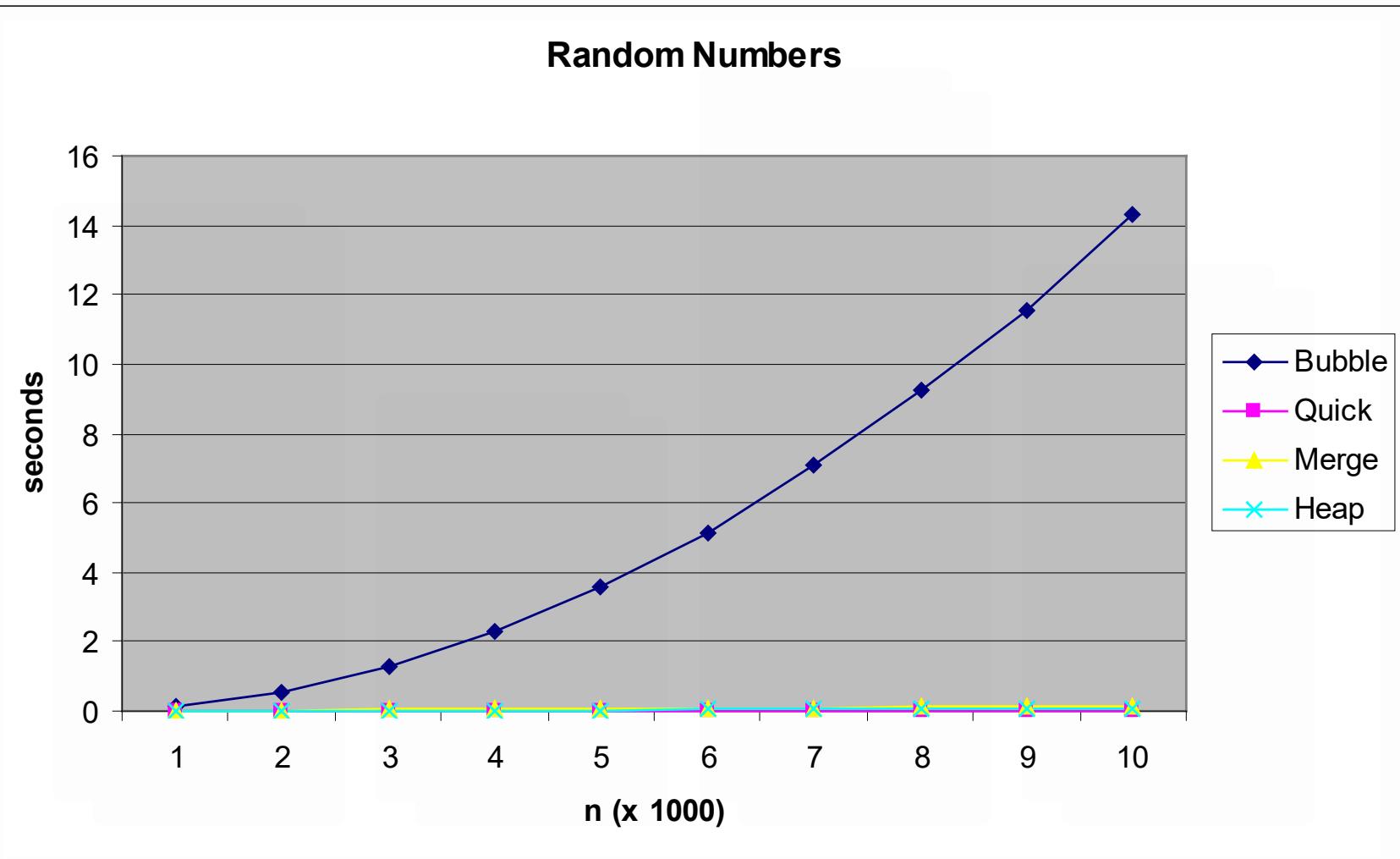
Data Structures and Abstractions

Empirical Comparisons, and the STL Sorts

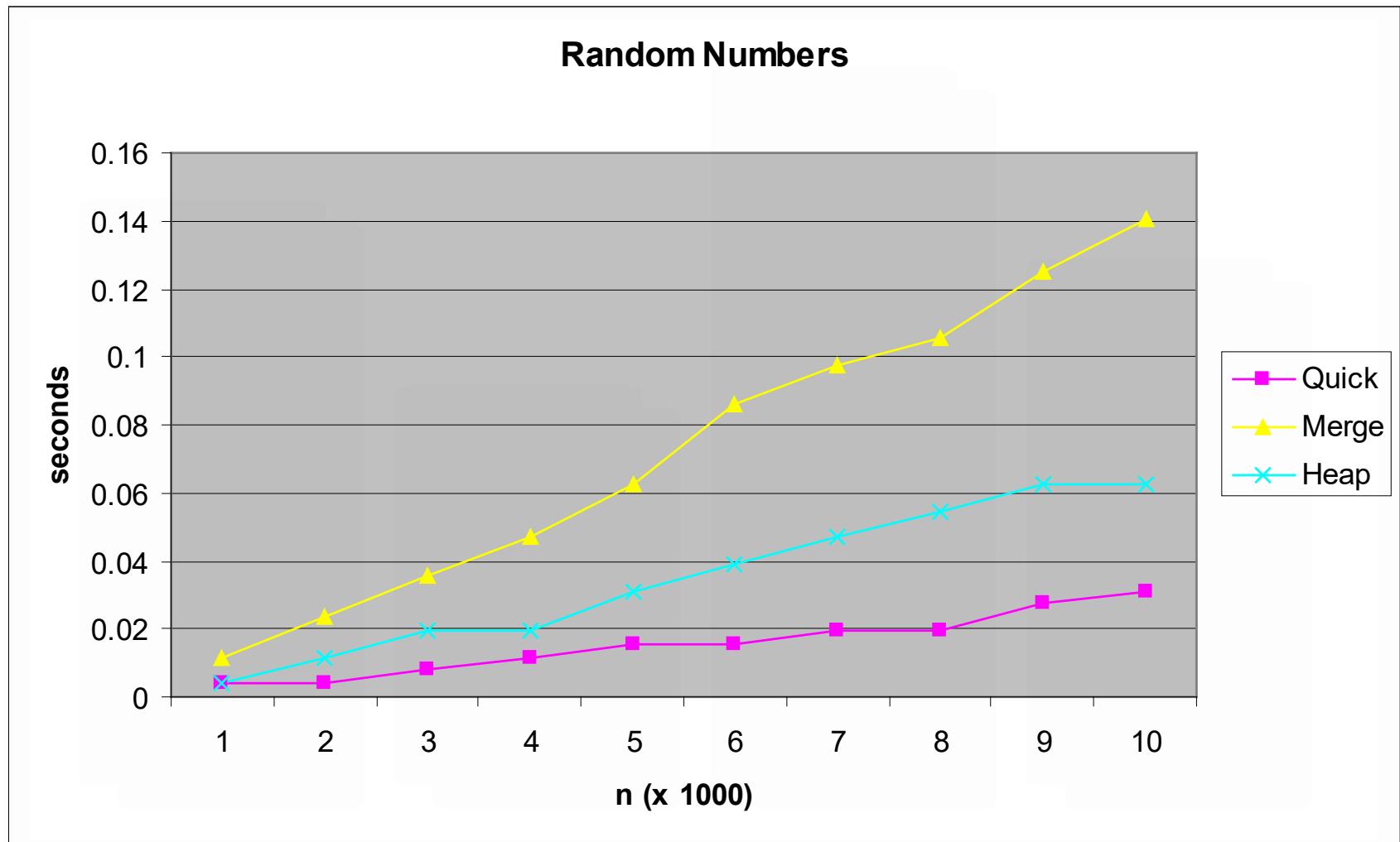
Lecture 27



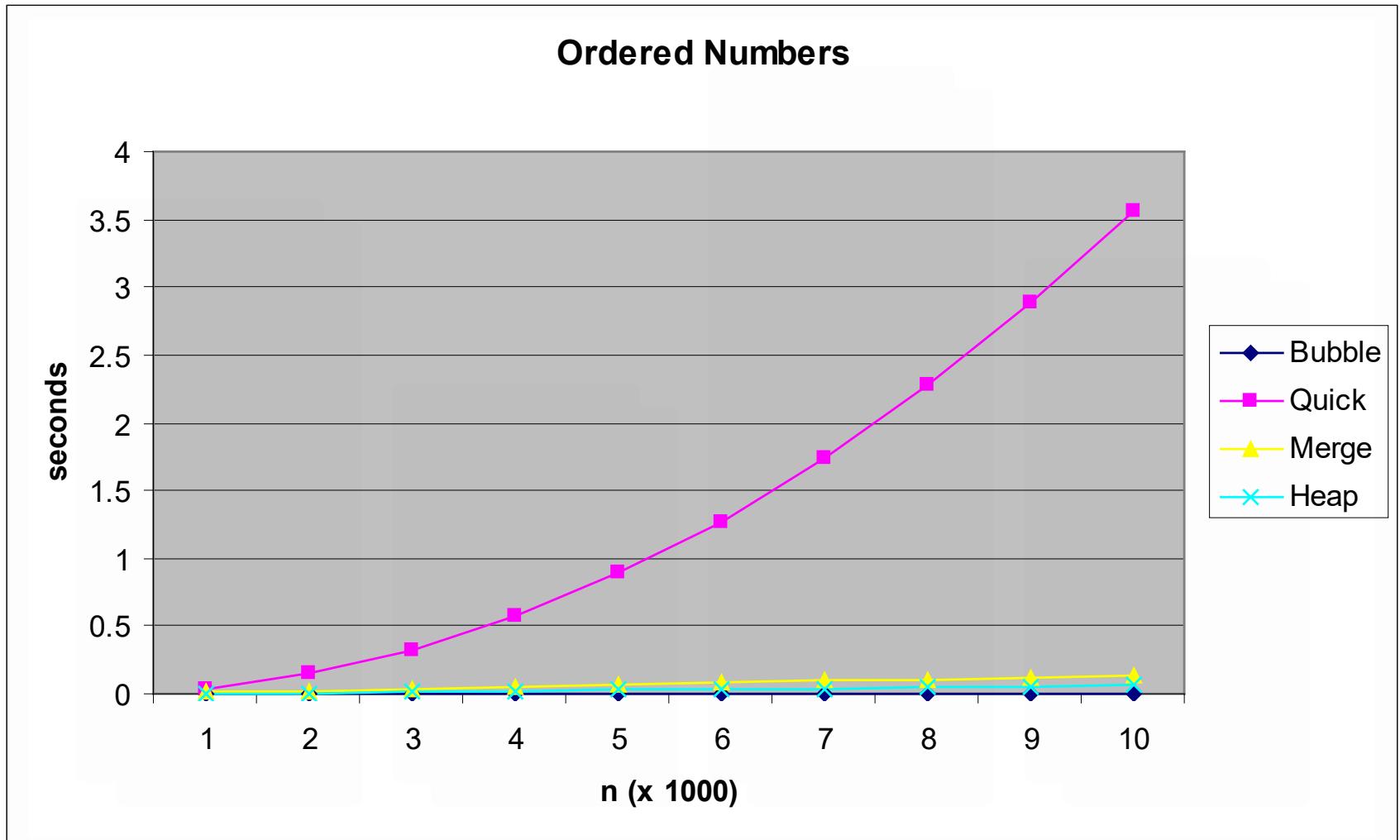
Empirical Comparison 1^[1]



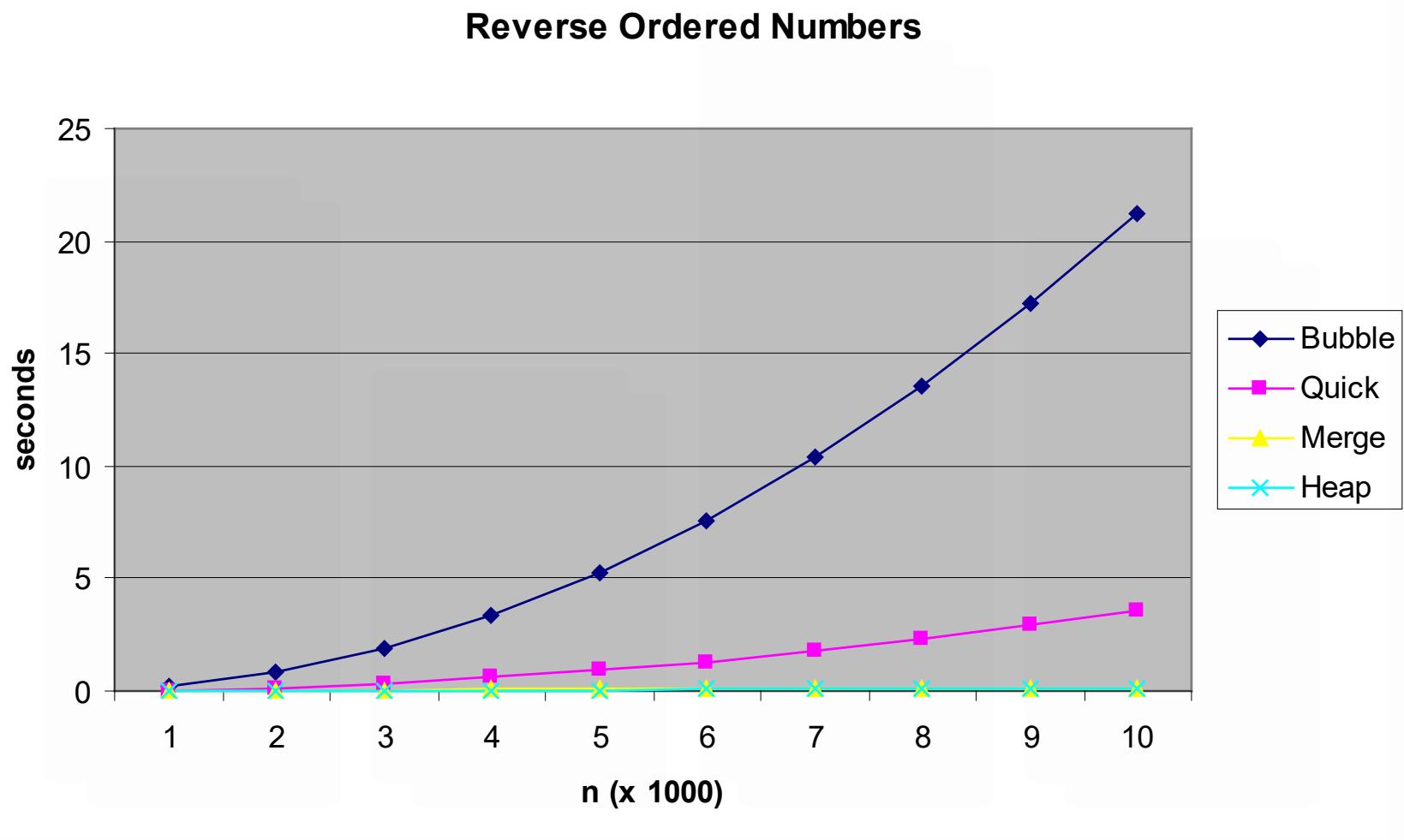
Empirical Comparison 2



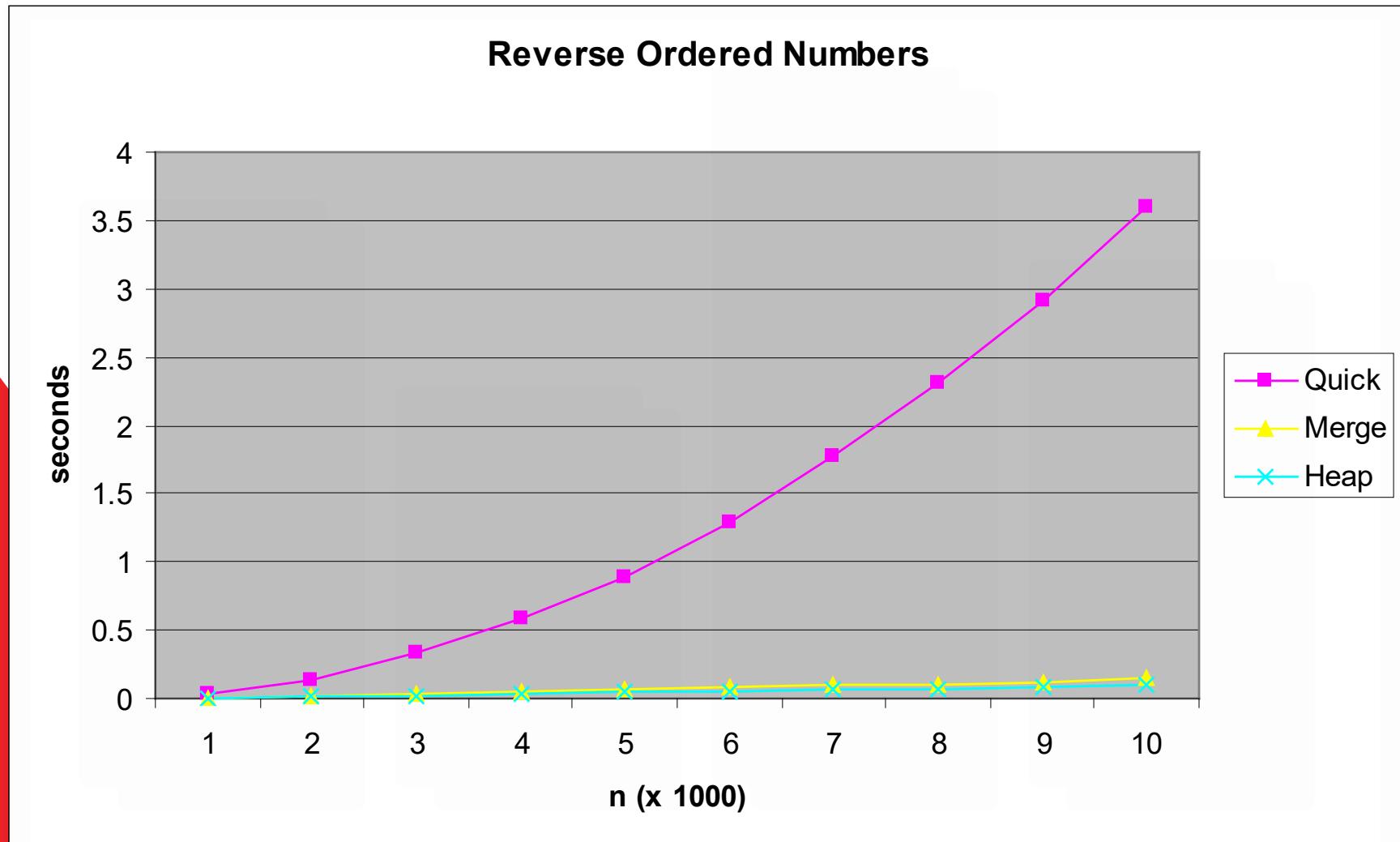
Empirical Comparison 3



Empirical Comparison 4



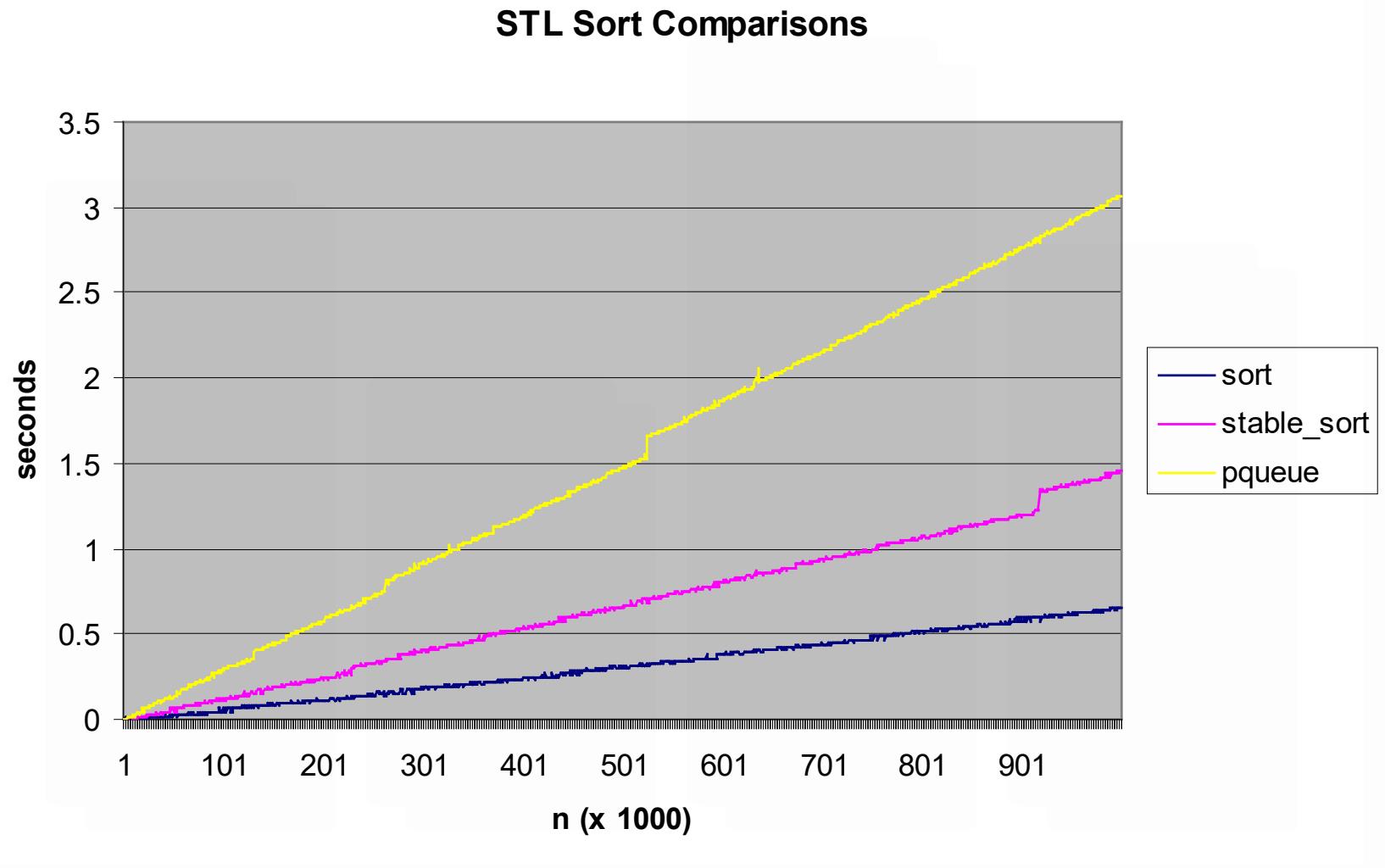
Empirical Comparison 5



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);
}
```

Readings

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



Murdoch
UNIVERSITY

Data Structures and Abstractions

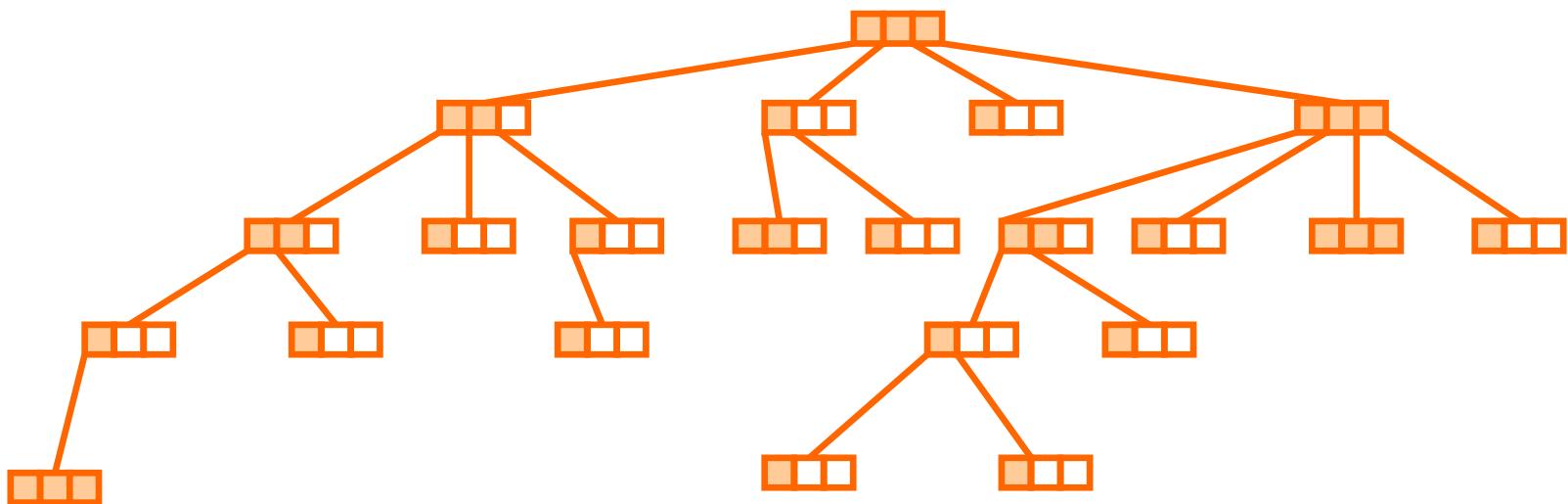
Trees and Tree Searching

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.

In-Order Traversal

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

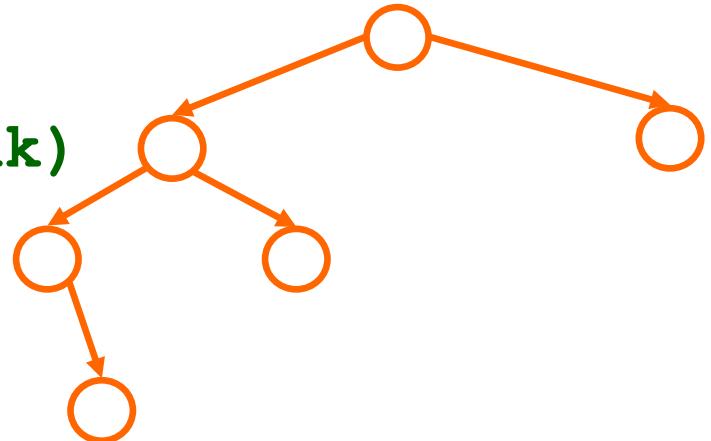
ProcessNode (node) [1]

ProcessNode (leftLink)

Process this node

ProcessNode (rightLink)

End ProcessNode



In-Order Traversal Animation

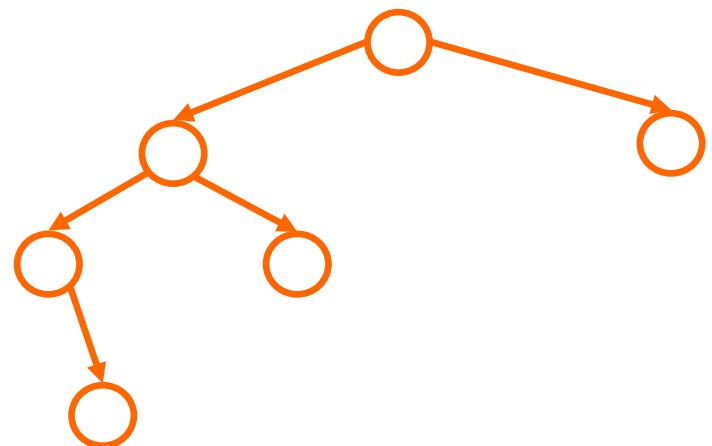
ProcessNode (node)

ProcessNode (leftLink)

Process this node

ProcessNode (rightLink)

End ProcessNode



Pre-Order Traversal

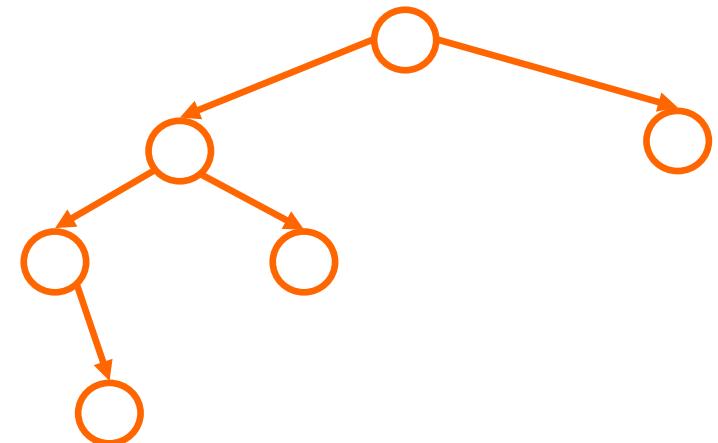
ProcessNode (node)

Process this node

ProcessNode (leftLink)

ProcessNode (rightLink)

End ProcessNode



Pre-Order Traversal Animation

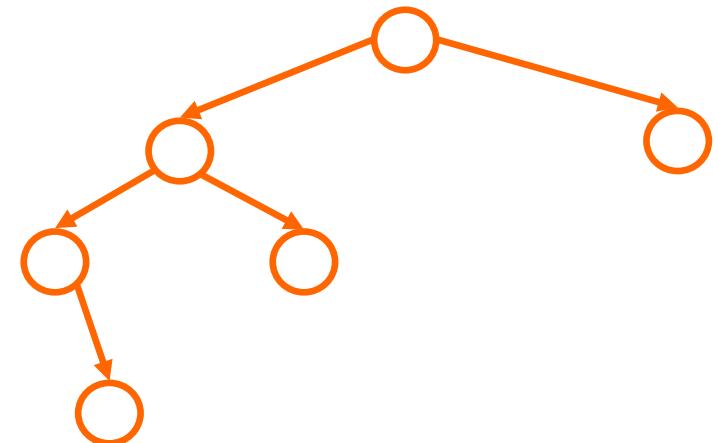
ProcessNode (node)

Process this node

ProcessNode (leftLink)

ProcessNode (rightLink)

End ProcessNode



Post-Order Traversal

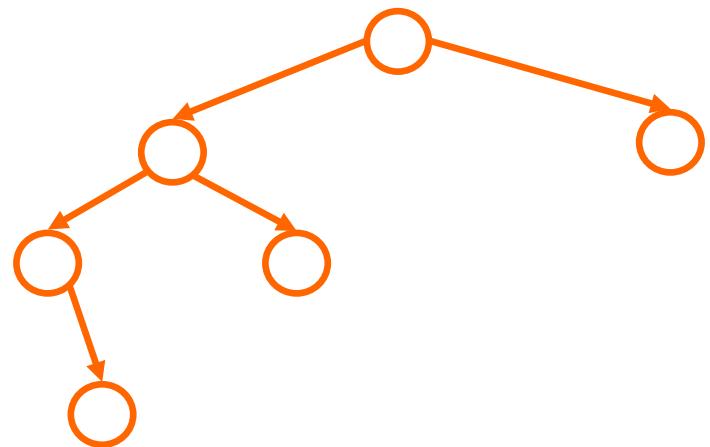
ProcessNode (node)

ProcessNode (leftLink)

ProcessNode (rightLink)

Process this node

End ProcessNode



Post-Order Traversal Animation

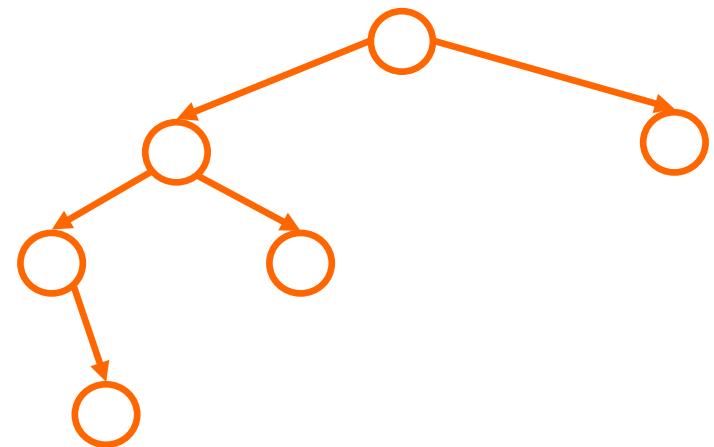
ProcessNode (node)

ProcessNode (leftLink)

ProcessNode (rightLink)

Process this node

End ProcessNode

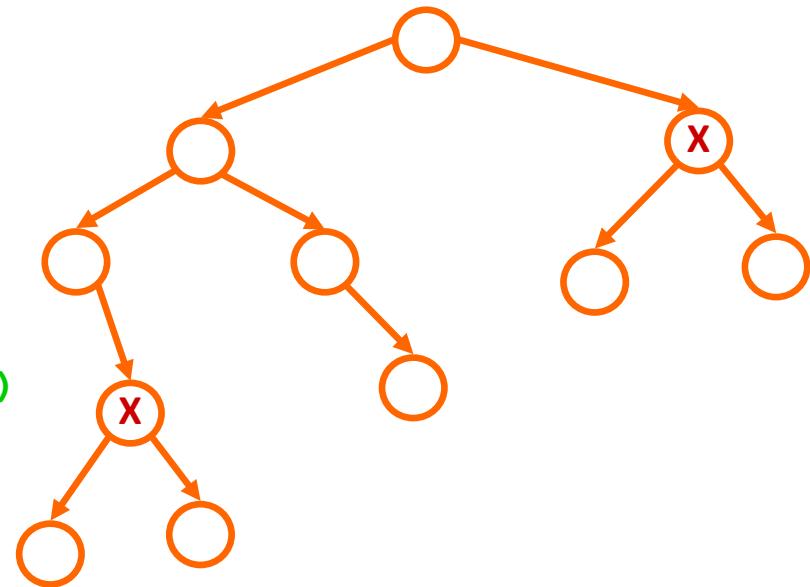


Tree Searching

- 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.

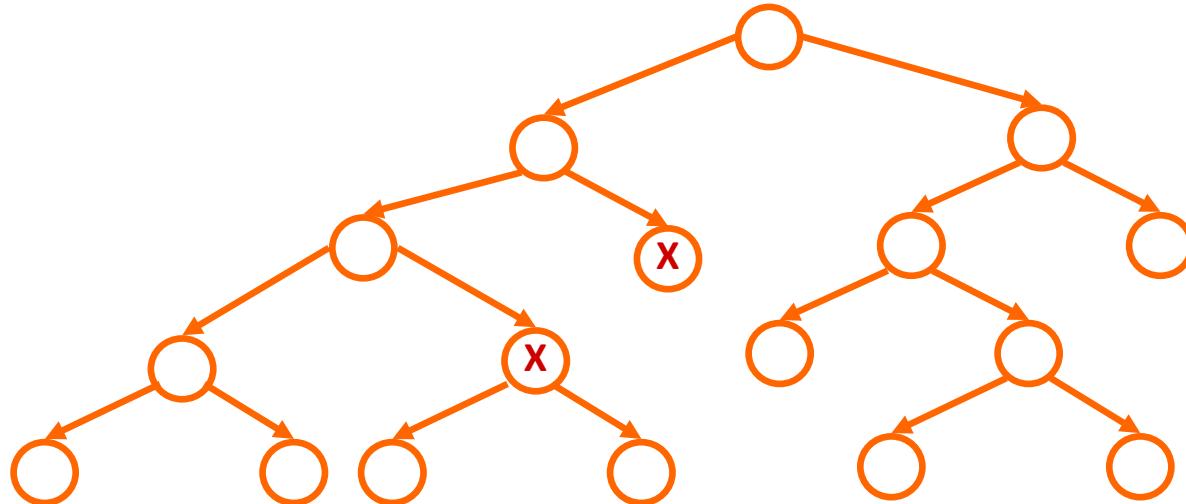
Depth First Search Animation

```
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
```

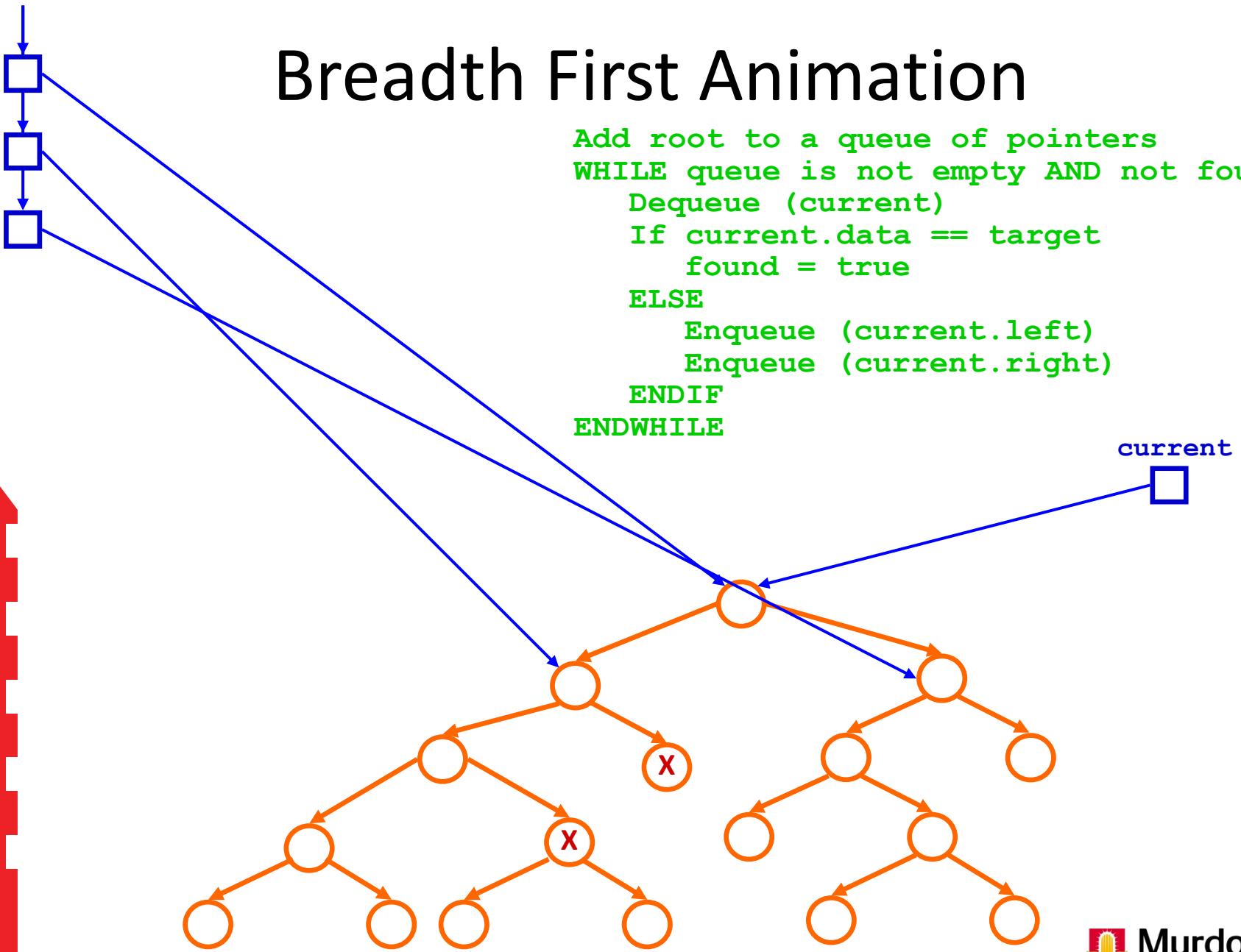


Breadth First Animation [1]

```
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



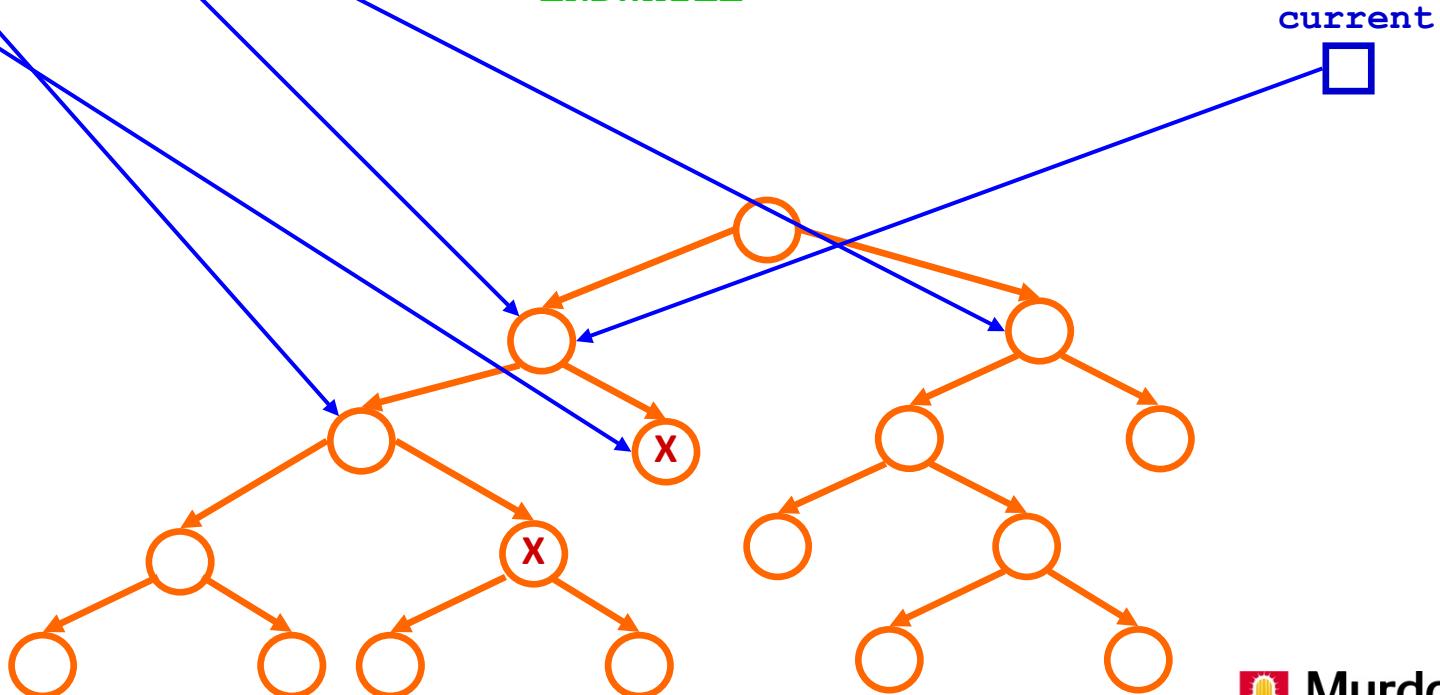
```

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

```

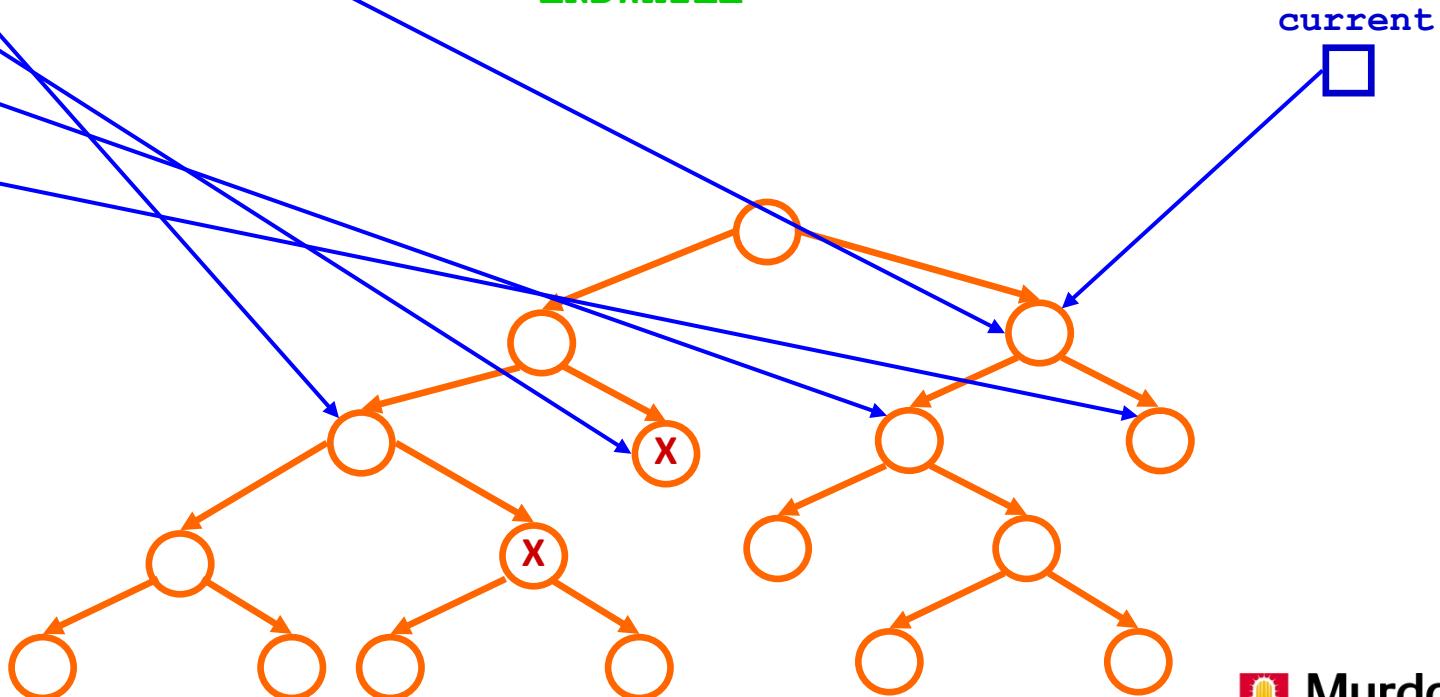
Breadth First Animation

```
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
```



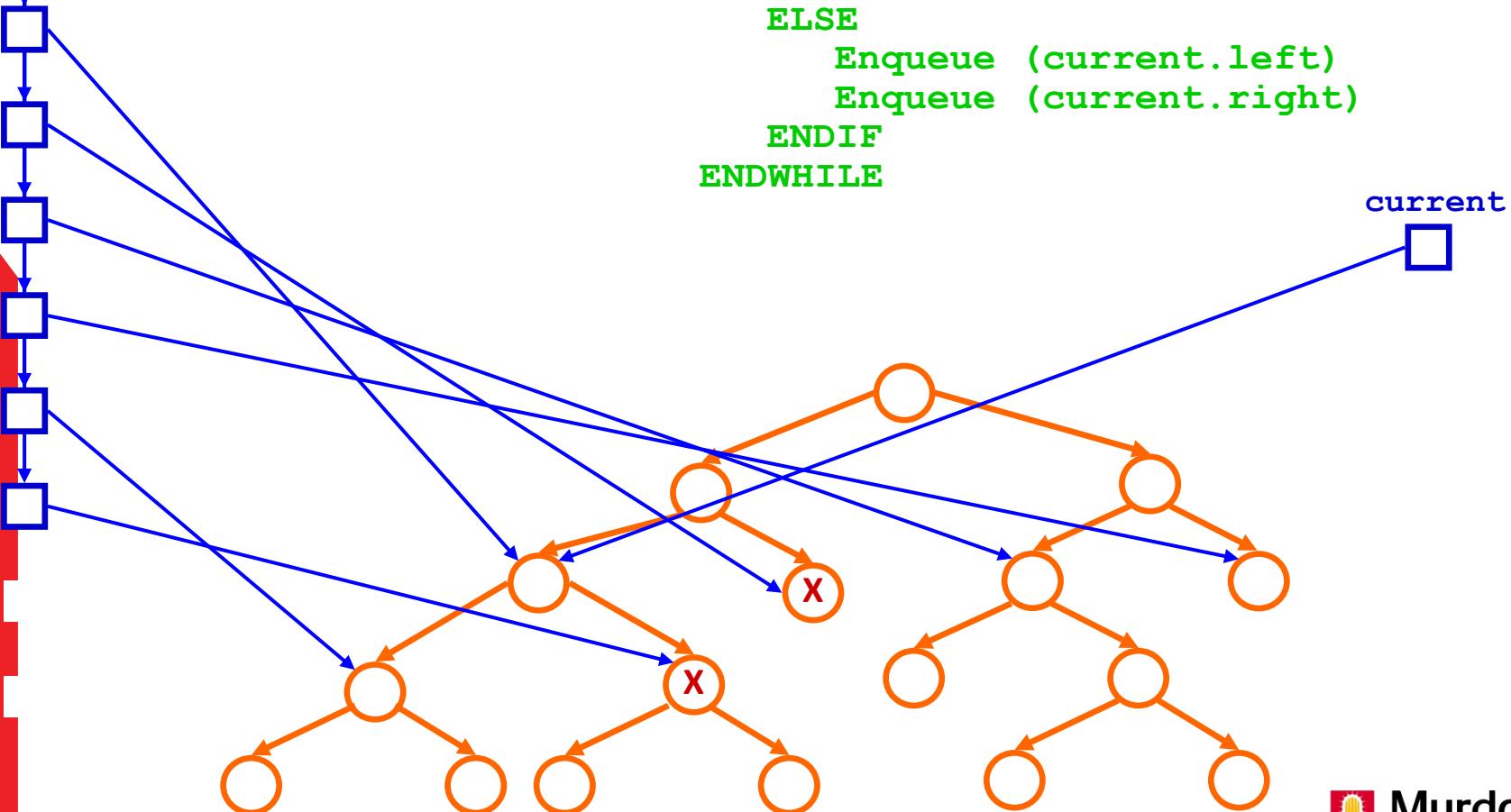
Breadth First Animation

```
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
```



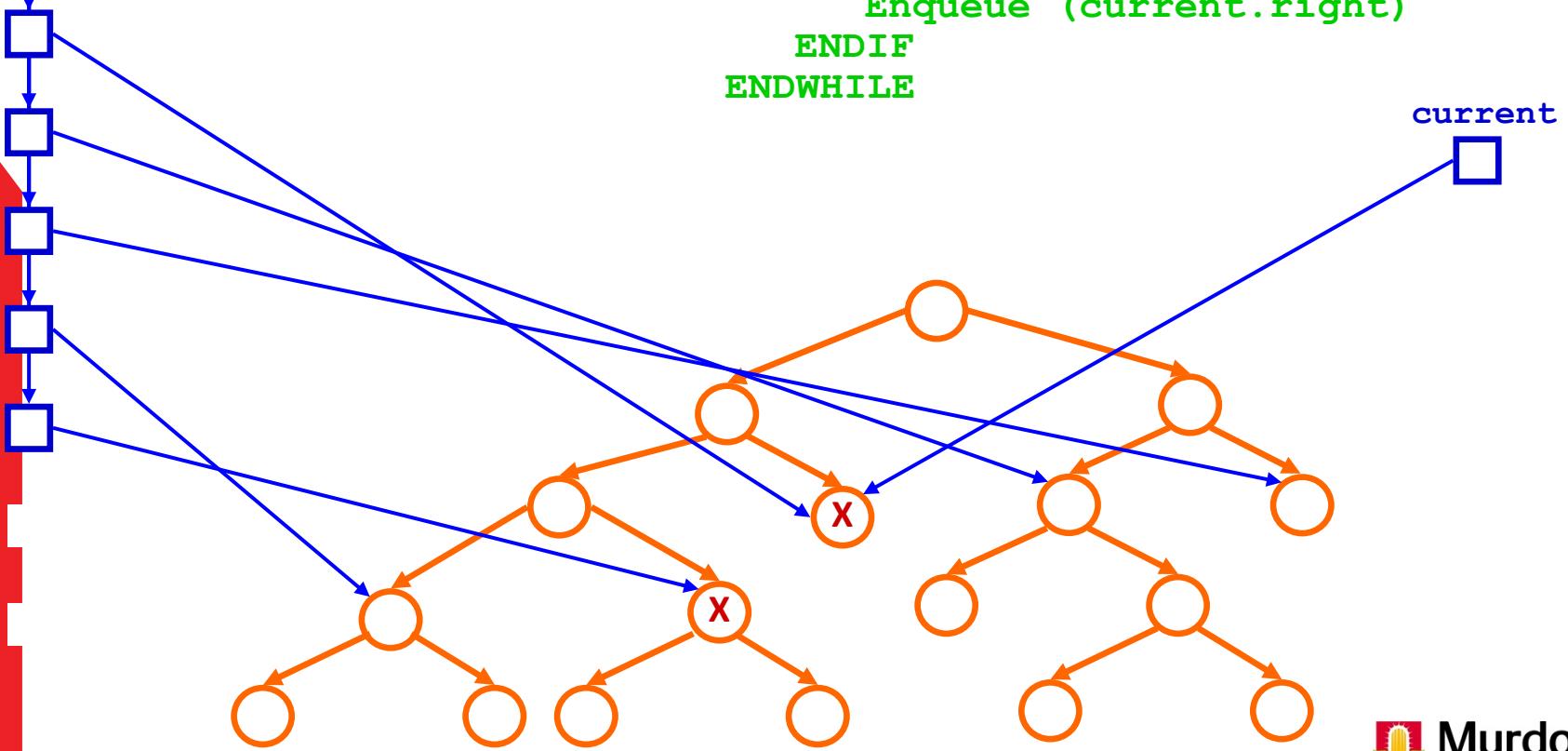
Breadth First Animation

```
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
```



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

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.

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 is also possible to have secondary keys, where a list, array or tree is ‘overlaid’ on the first structure giving a different sorted order.

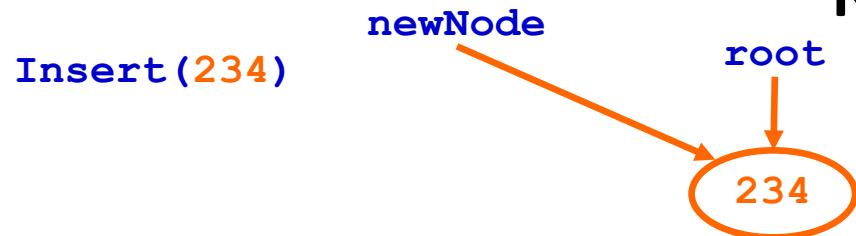
Binary Search Trees

- 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

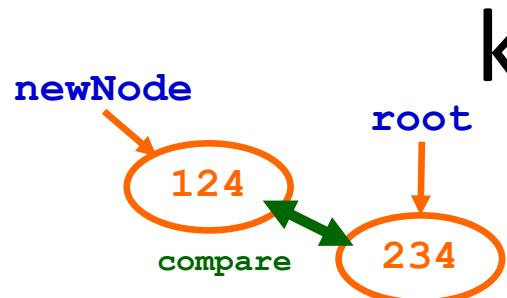
- Almost all BST algorithms are recursive as this makes them very simple.
- However, the root node might be treated differently because it has no parent but should it? [1]
- All the methods require that root has been set to NULL in the constructor.
- Traversal of a BST is almost always done either in-order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

BST Insert Animation (for integer key)



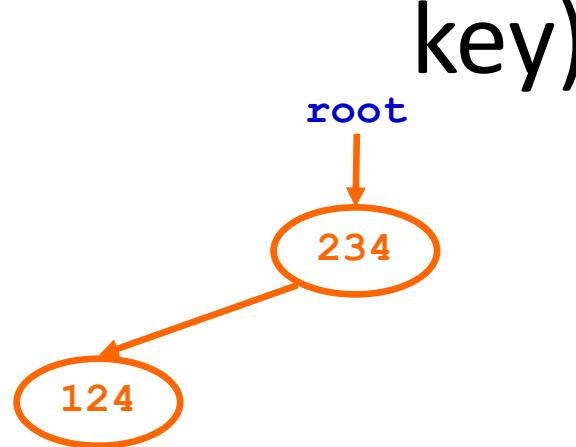
BST Insert Animation (for integer key)

Insert(124)



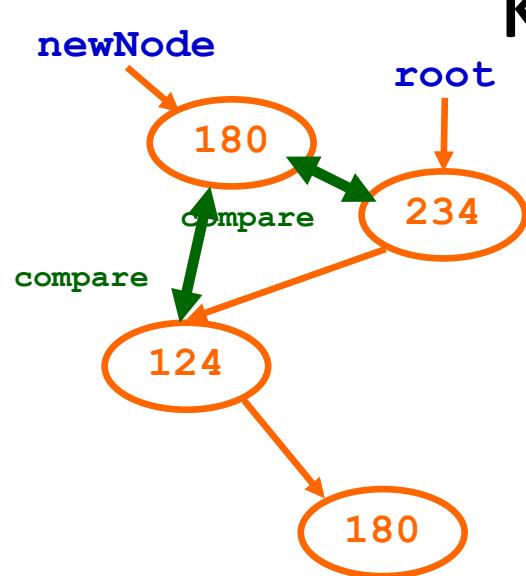
BST Insert Animation (for integer key)

Insert(124)



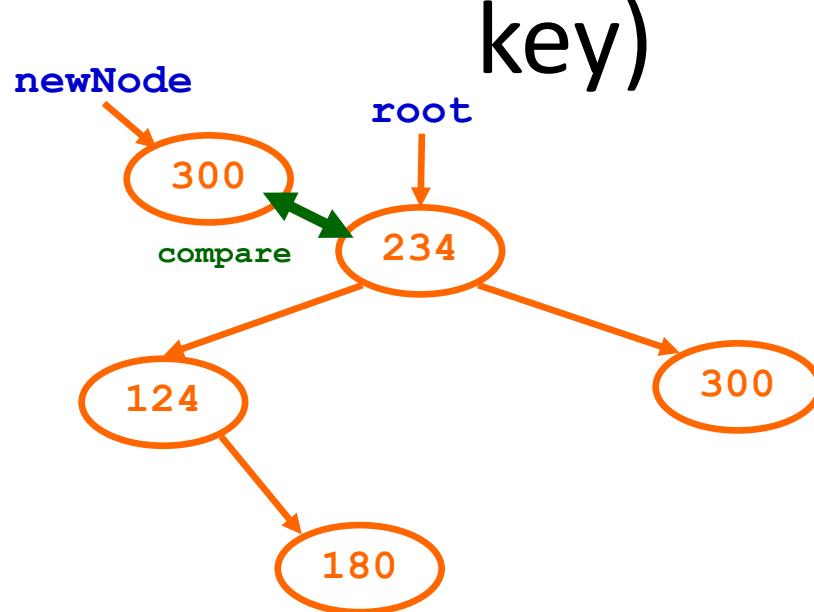
BST Insert Animation (for integer key)

Insert(180)



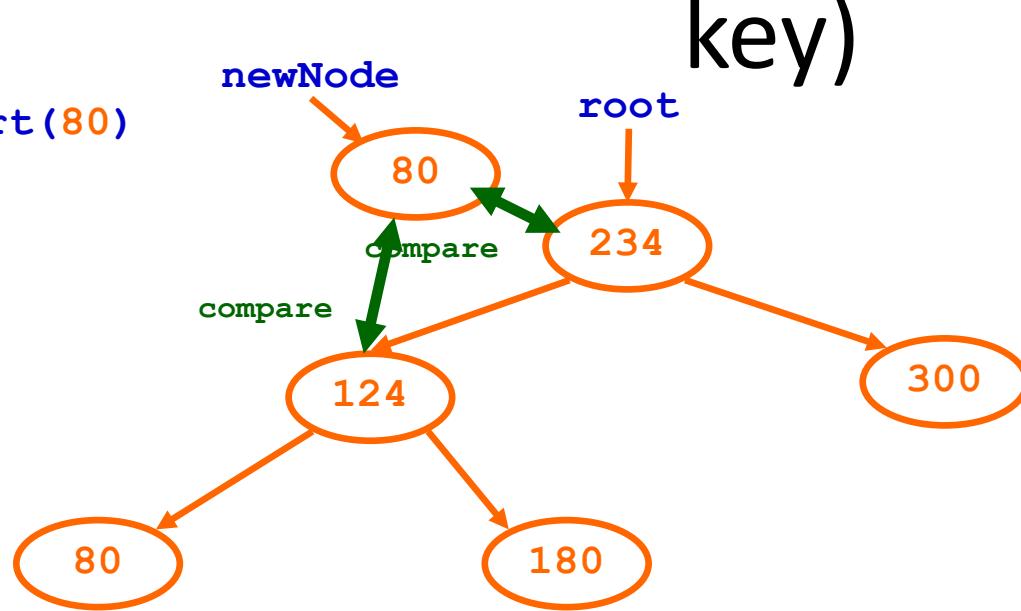
BST Insert Animation (for integer key)

Insert(300)



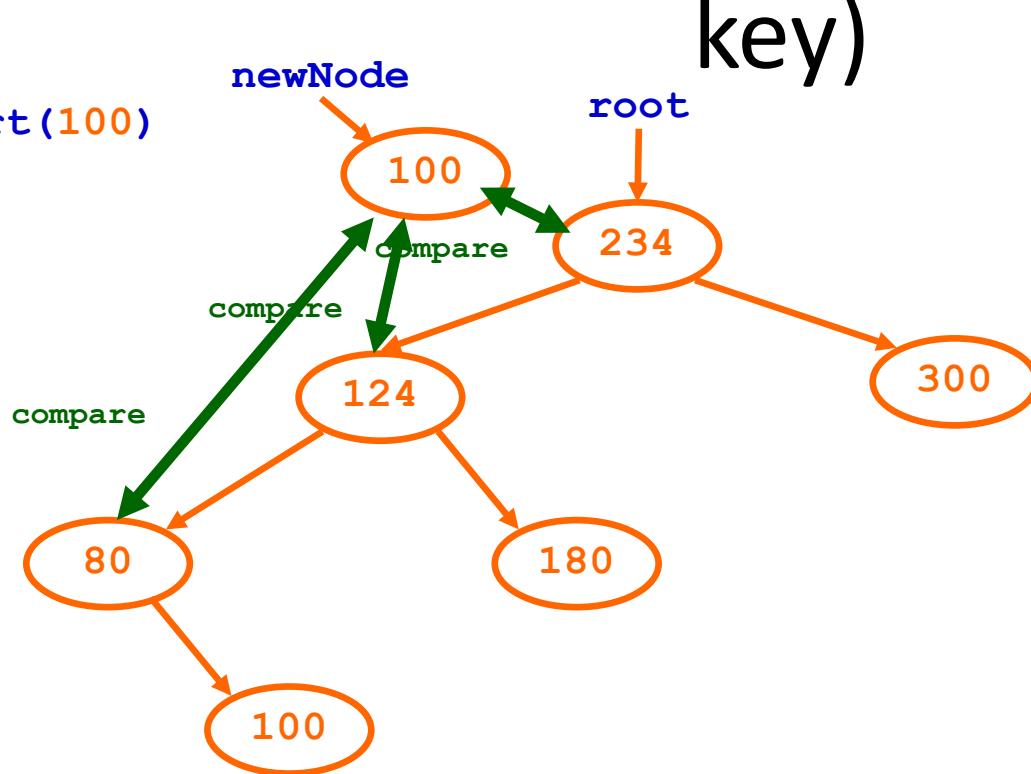
BST Insert Animation (for integer key)

Insert(80)



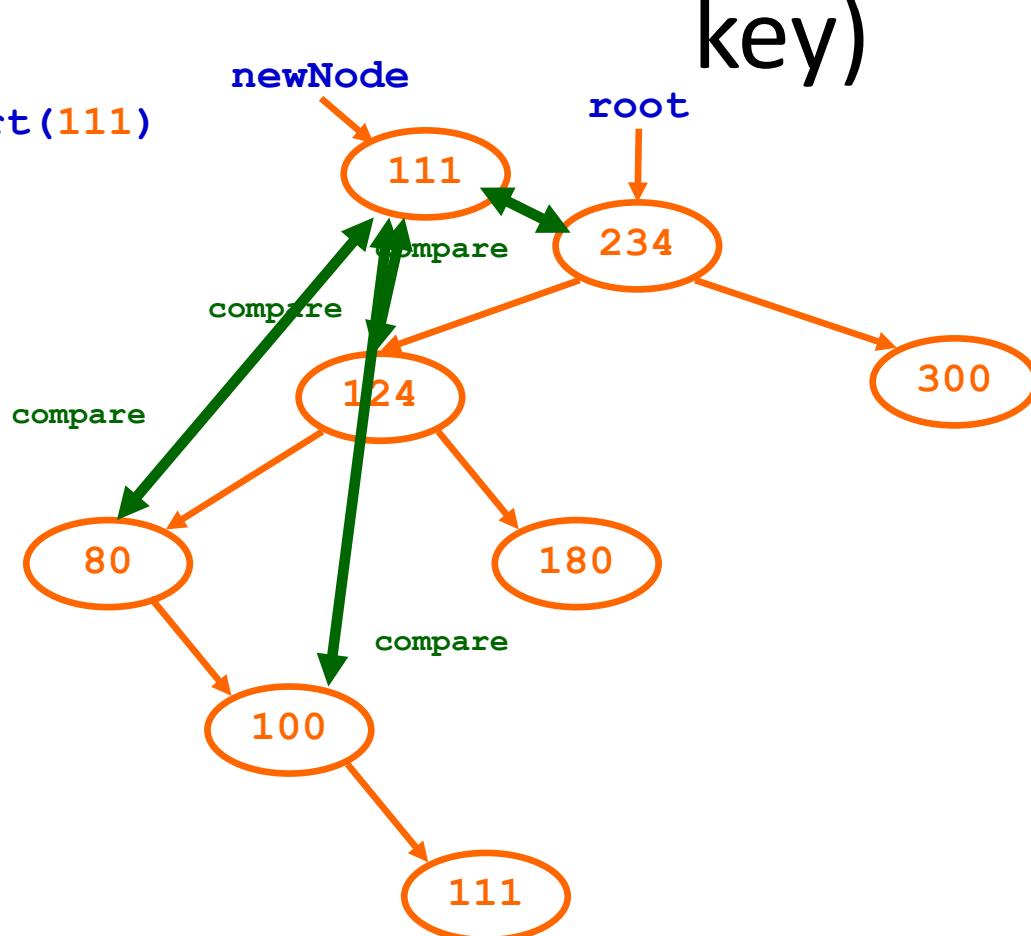
BST Insert Animation (for integer key)

Insert(100)



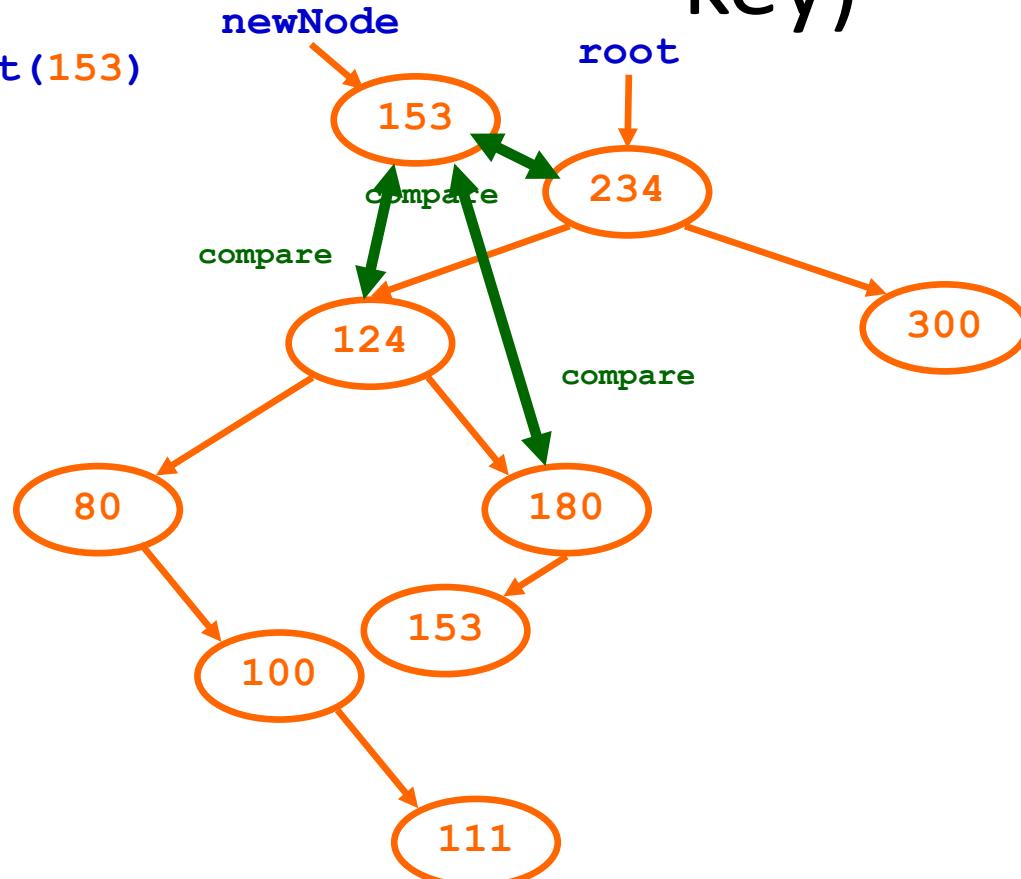
BST Insert Animation (for integer key)

Insert(111)



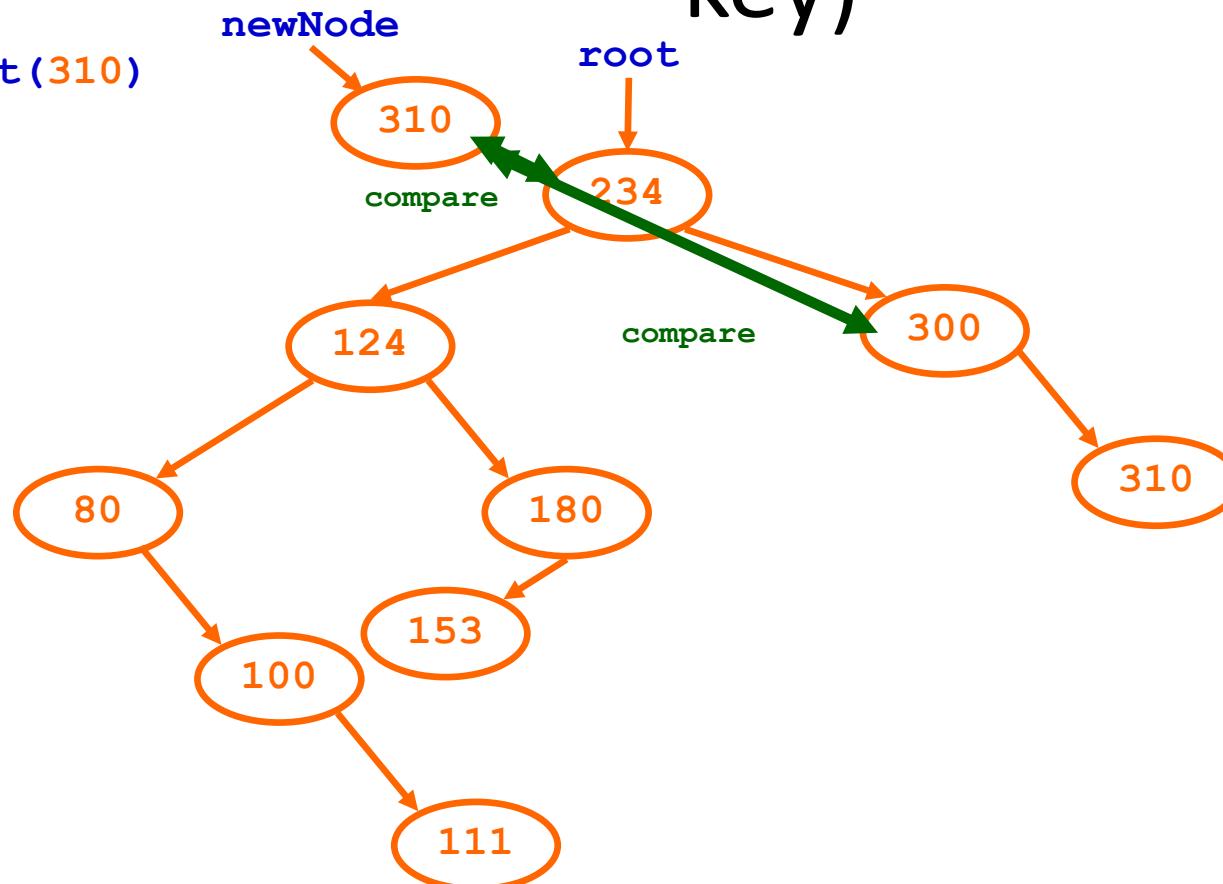
BST Insert Animation (for integer key)

Insert(153)



BST Insert Animation (for integer key)

Insert(310)



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

- 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

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.

AVL Trees

- Invented by **Adelson-Velski** and **Landis** (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:

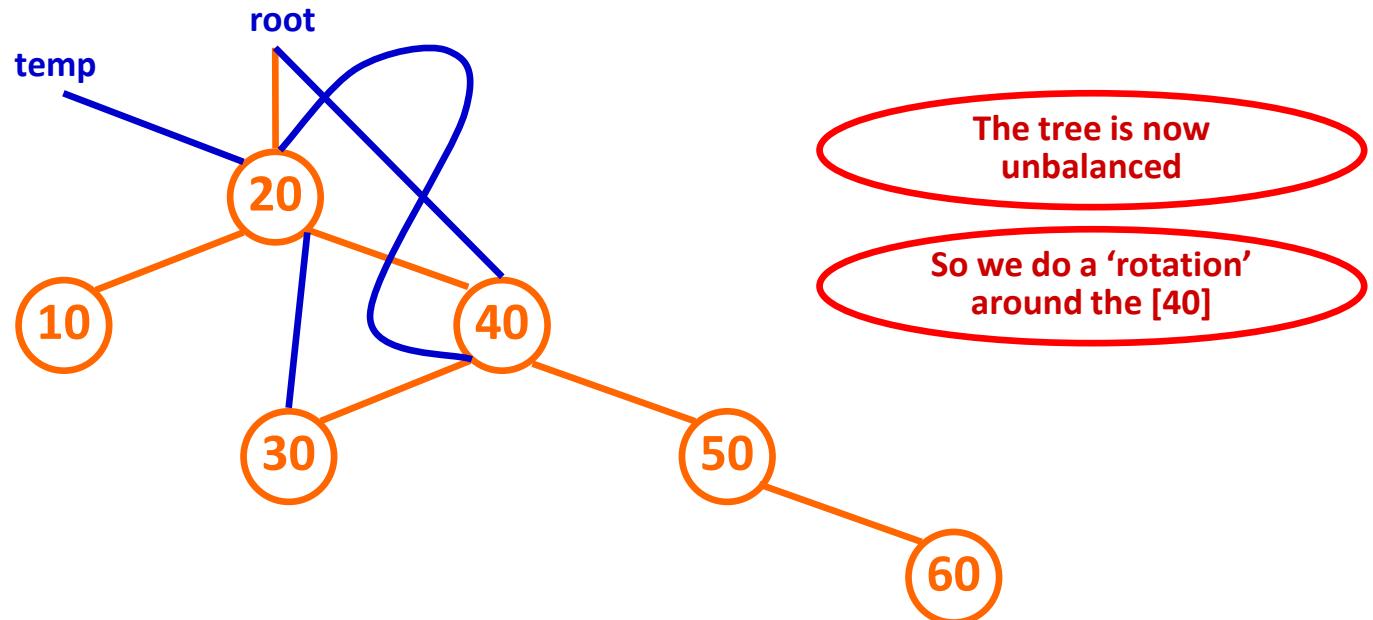
```
IF the tree is empty
    height = -1
ELSE
    height = 1 + max(height(leftLink) , height(rightLink))
ENDIF
```

Insertion into an AVL Tree

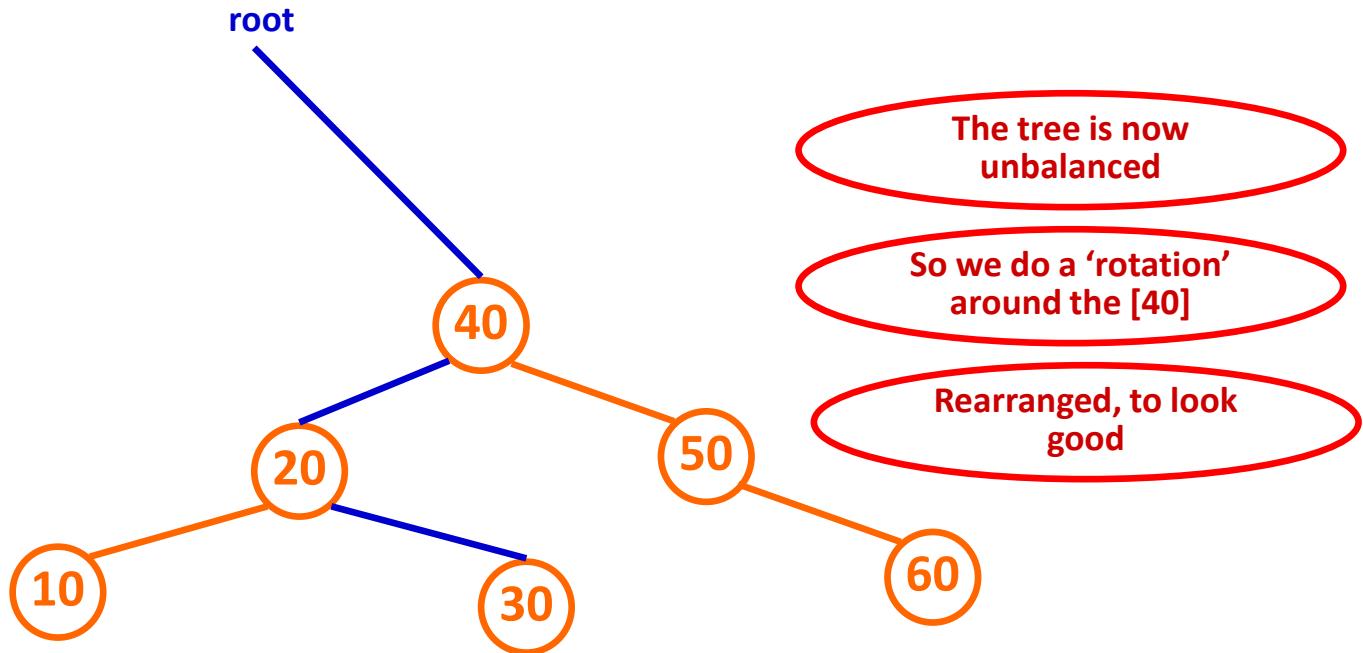
- Insertion is done the same as for an ordinary BST.
- But if the height is unbalanced, the insertion is followed with a rebalance:

```
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
```

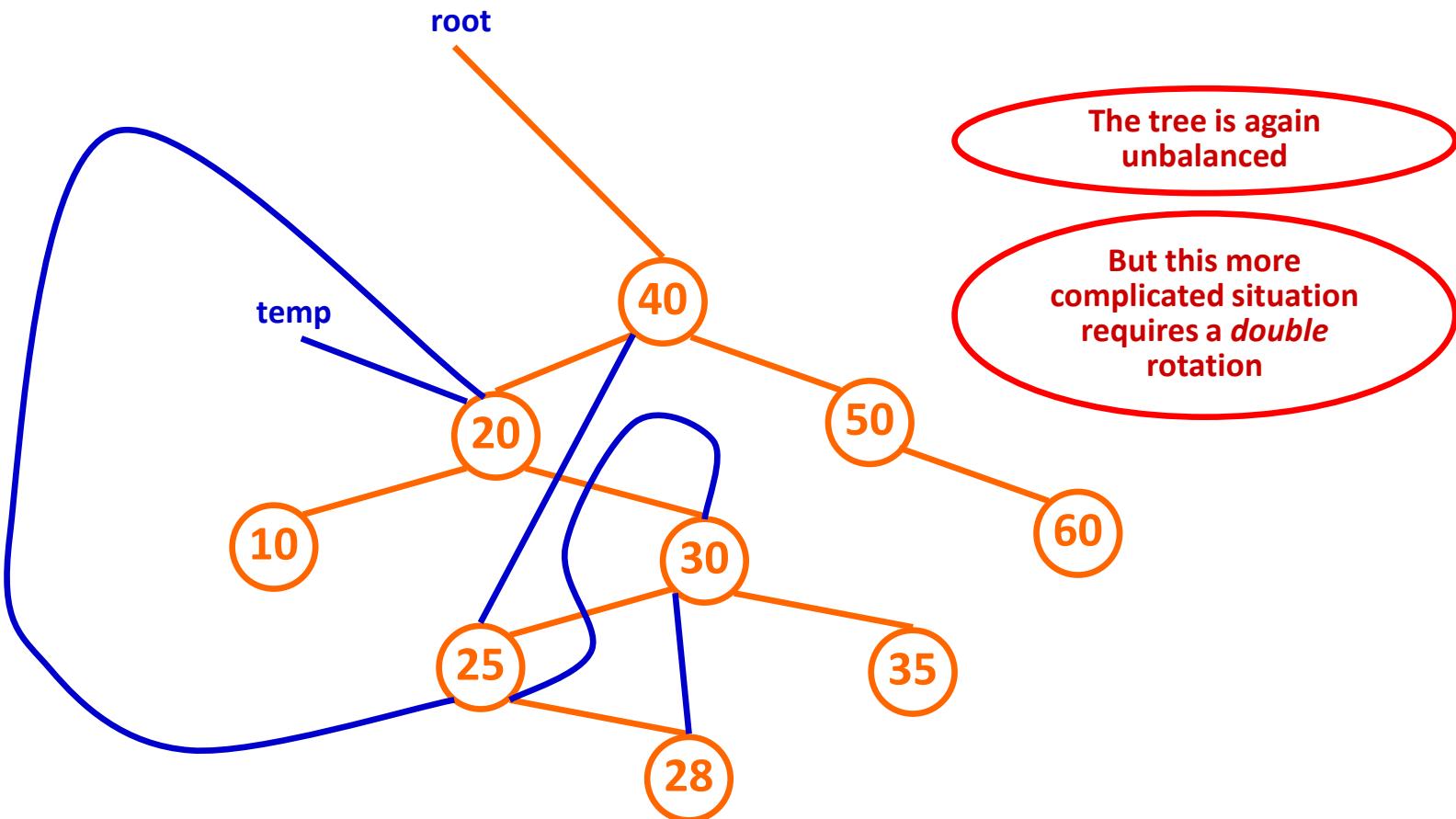
AVL Tree Insert Animation



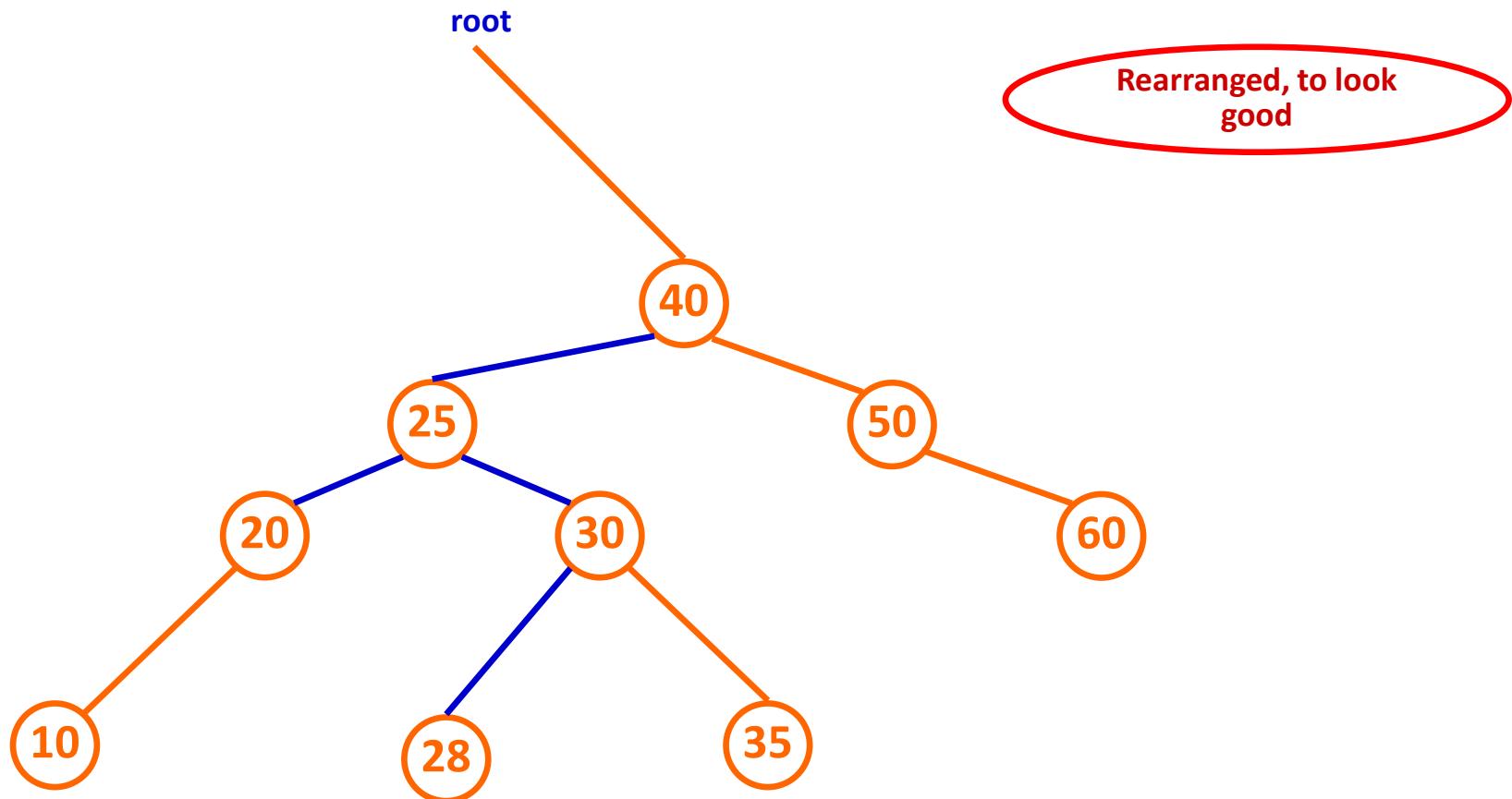
AVL Tree Insert Animation



AVL Tree Insert Animation



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).

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=PLUI4u3cNGP61Oq3tWYp6V_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.

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=8F2OvQlIGiU>
- 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=PLUI4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb
 - MIT Tutorial (different to a lab, no computers)
https://www.youtube.com/watch?v=lWzYoXKaRlc&list=PLUI4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb&index=29
 - https://www.youtube.com/watch?v=r5pXu1PAUkl&list=PLUI4u3cNGP61Oq3tWYp6V_F-5jb5L2iHb&index=28



Murdoch
UNIVERSITY

Data Structures and Abstractions

Multiway Trees

Lecture 30

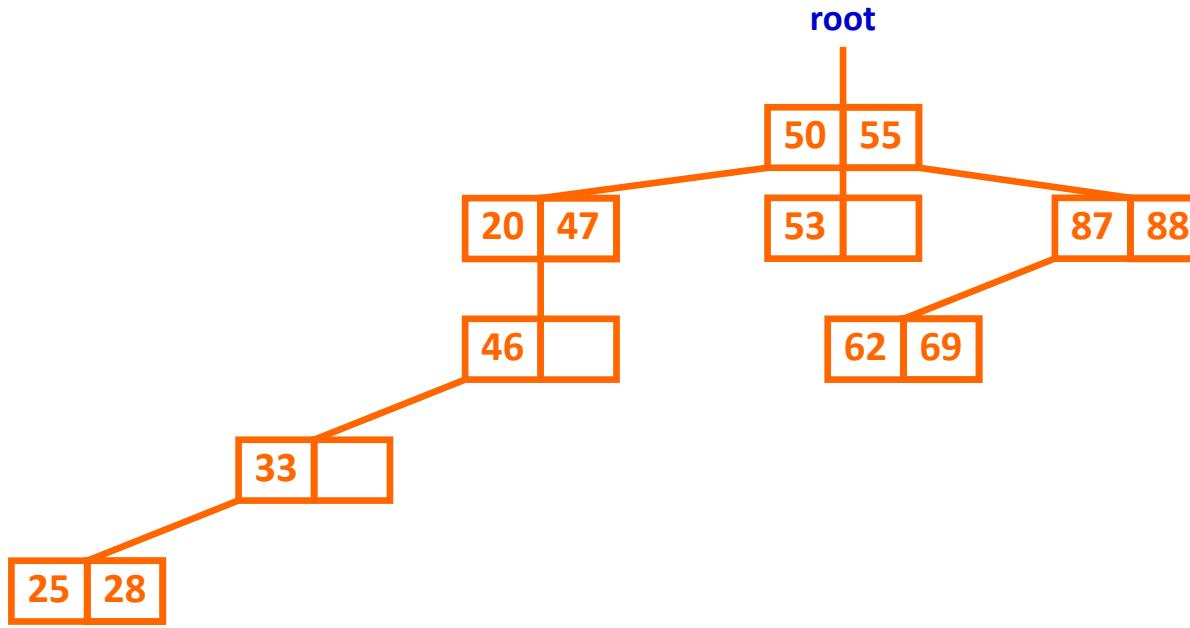


Multiway Trees

- 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]

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)



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

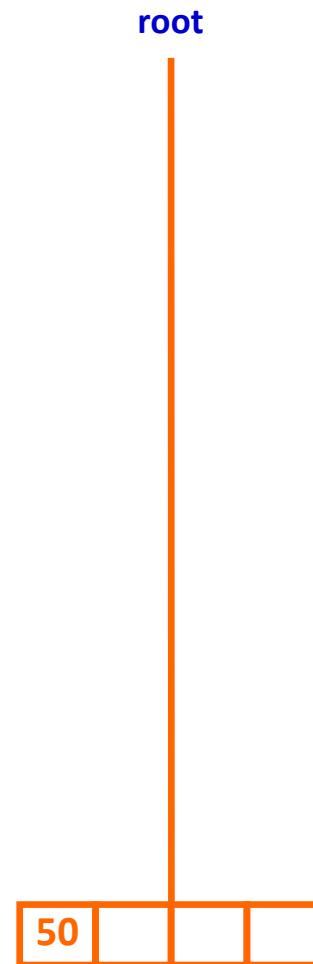
- Formally, a B Tree of order m is a multi-way tree in which:
 - the root is either a leaf or has at least two subtrees;
 - each leaf node holds at least $m/2$ keys;
 - each non-leaf node holds $k-1$ keys and k pointers to subtrees where $m/2 \leq k \leq m$;
 - all leaves are on the same level.
- m is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

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 is 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.

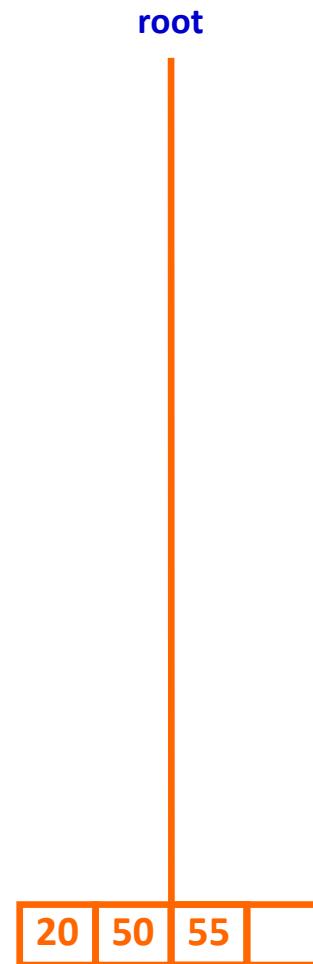
Insert(50)
Insert(20)

5-way B Tree Insert Animation



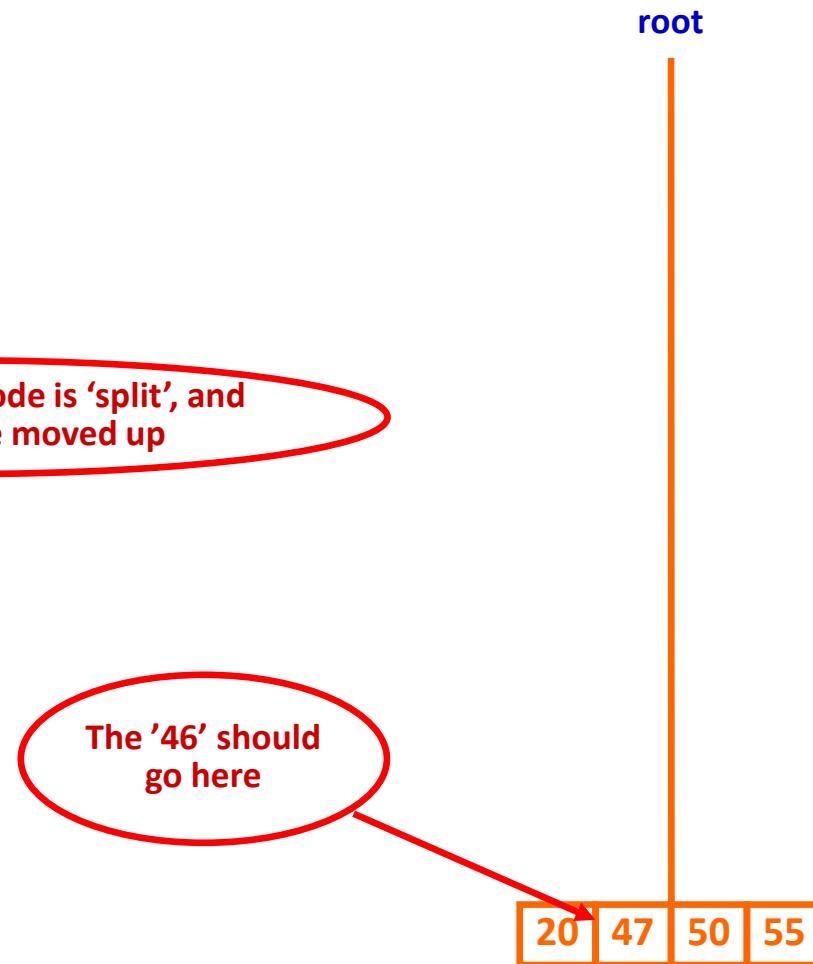
Insert(50)
Insert(20)
Insert(55)
Insert(47)

5-way B Tree Insert Animation



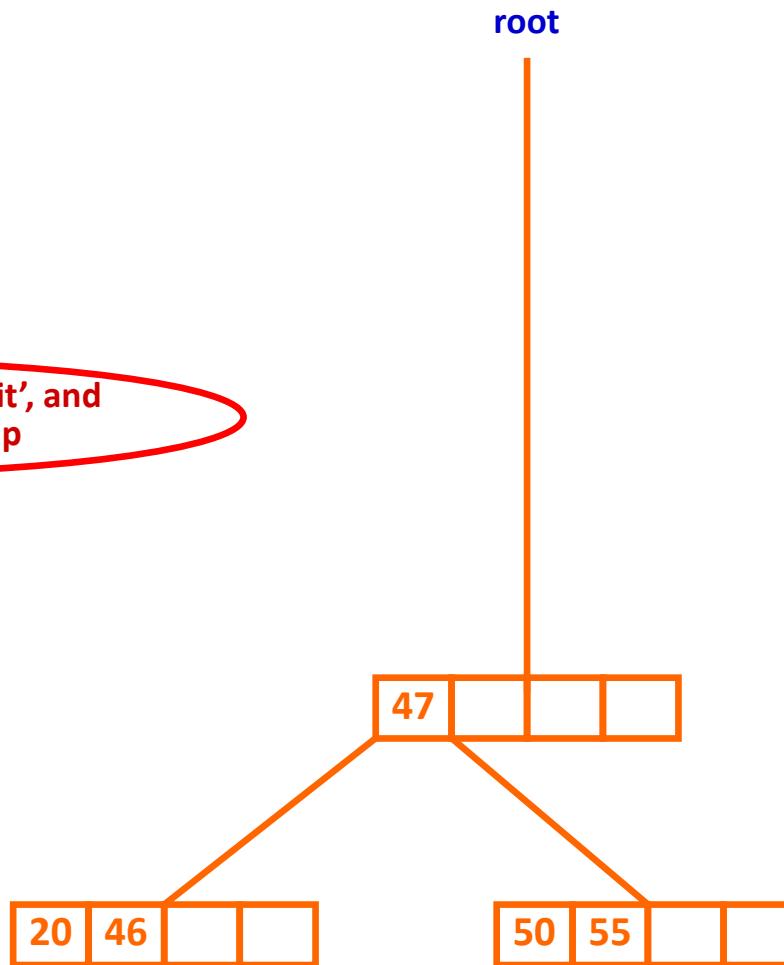
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)

5-way B Tree Insert Animation



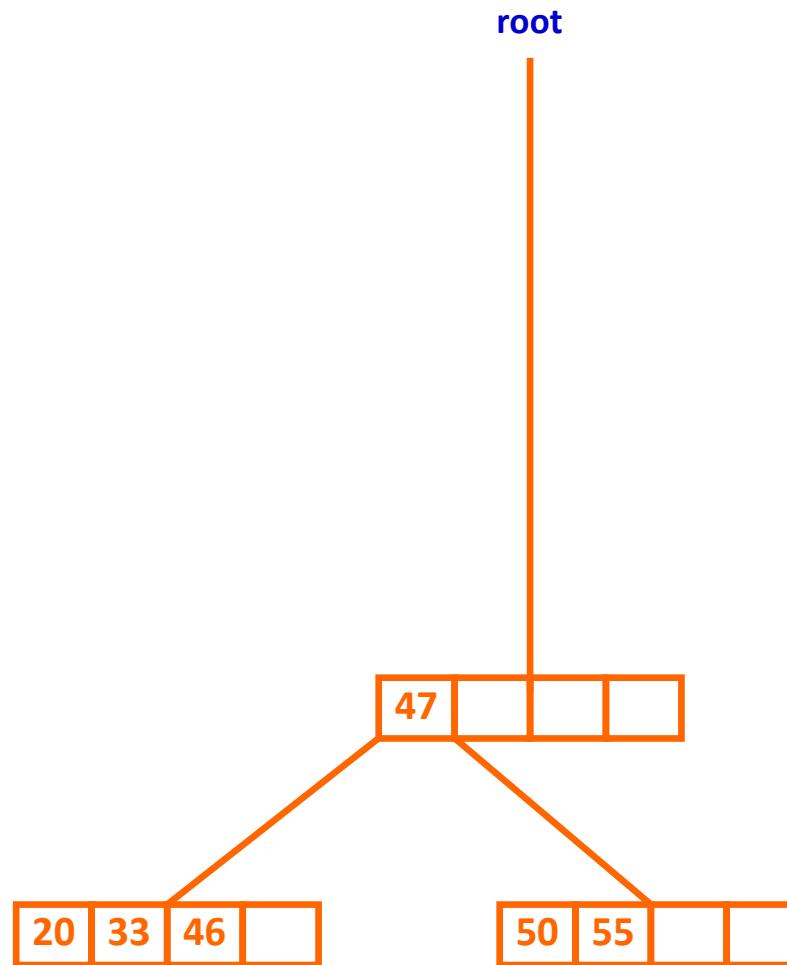
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)

5-way B Tree Insert Animation



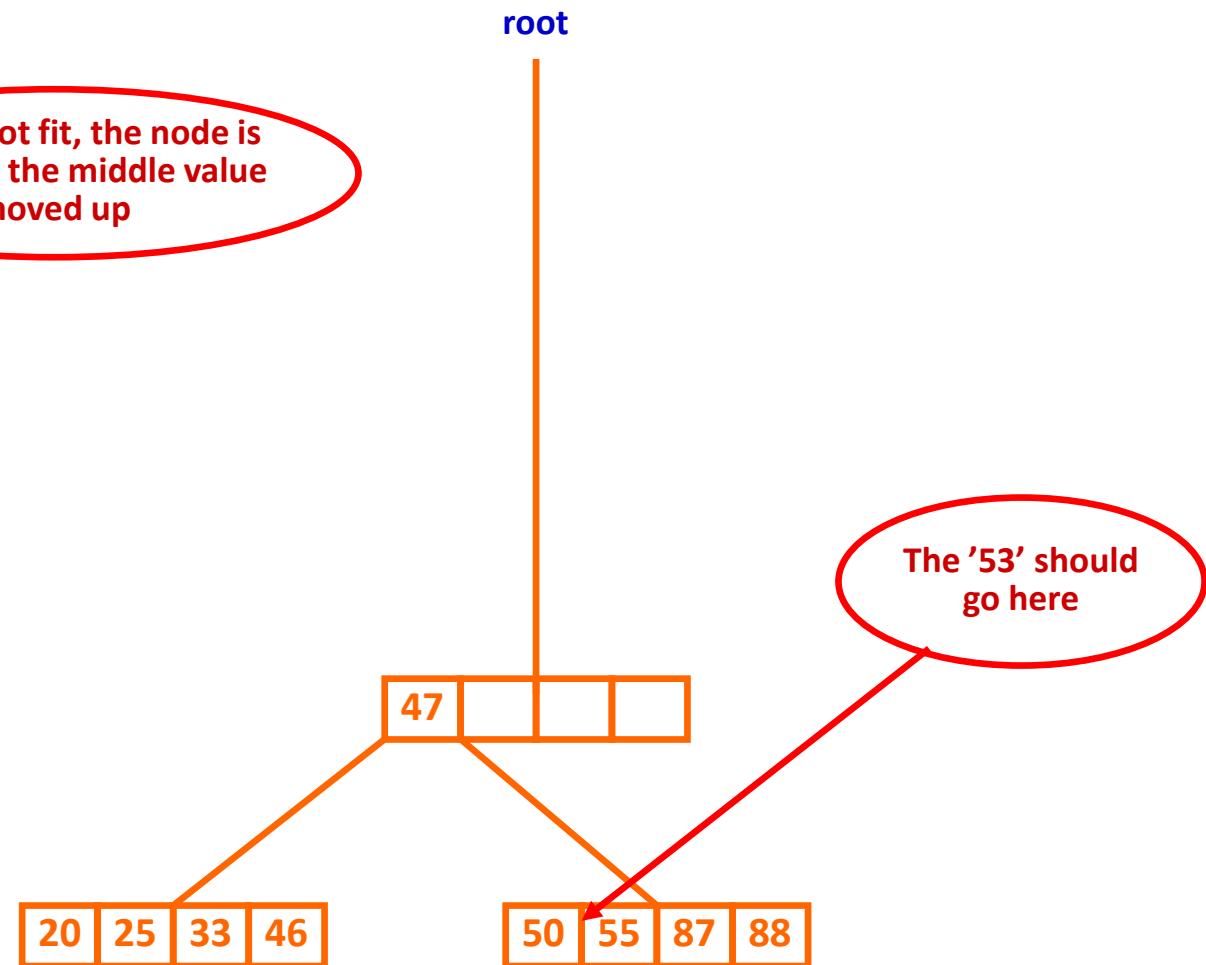
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)

5-way B Tree Insert Animation



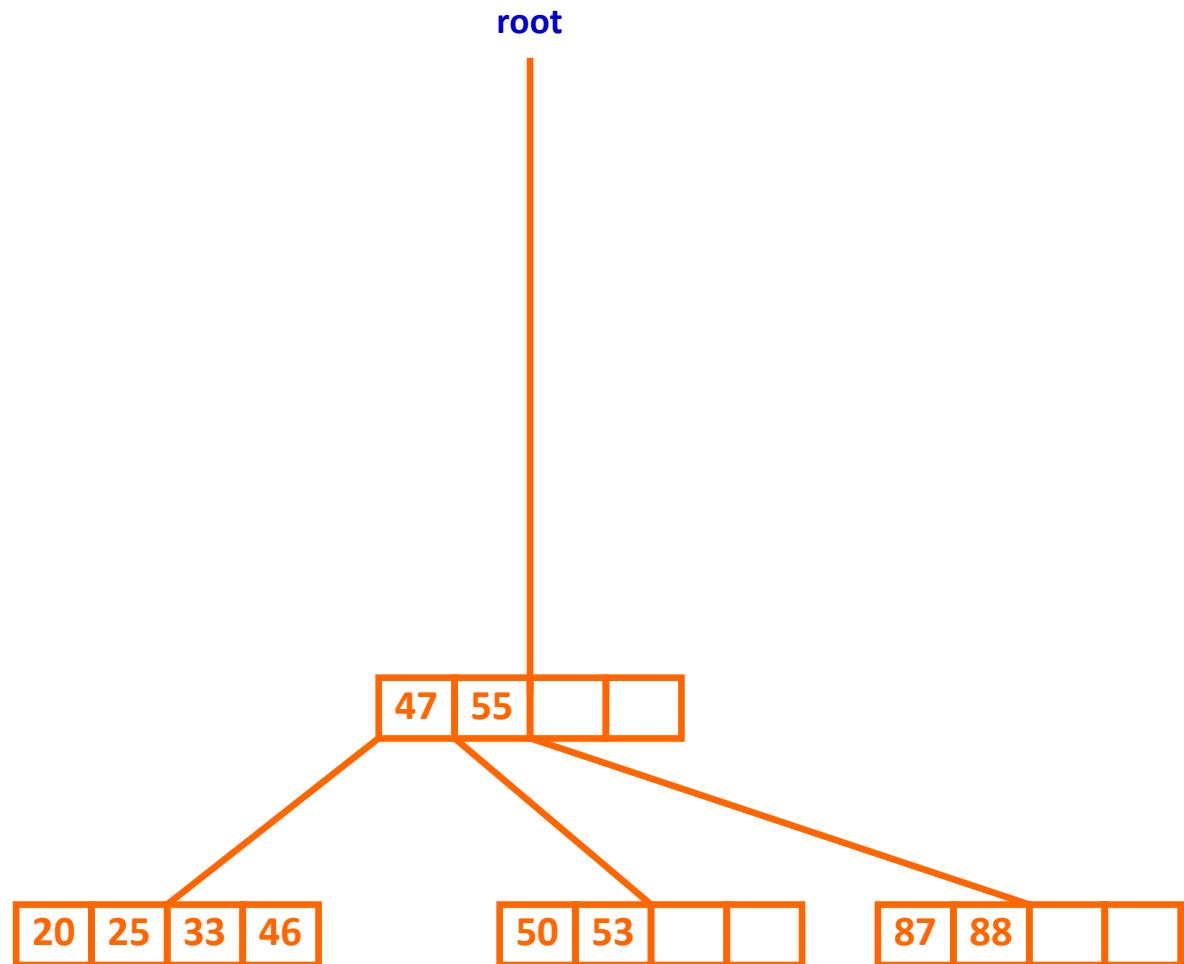
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)

5-way B Tree Insert Animation



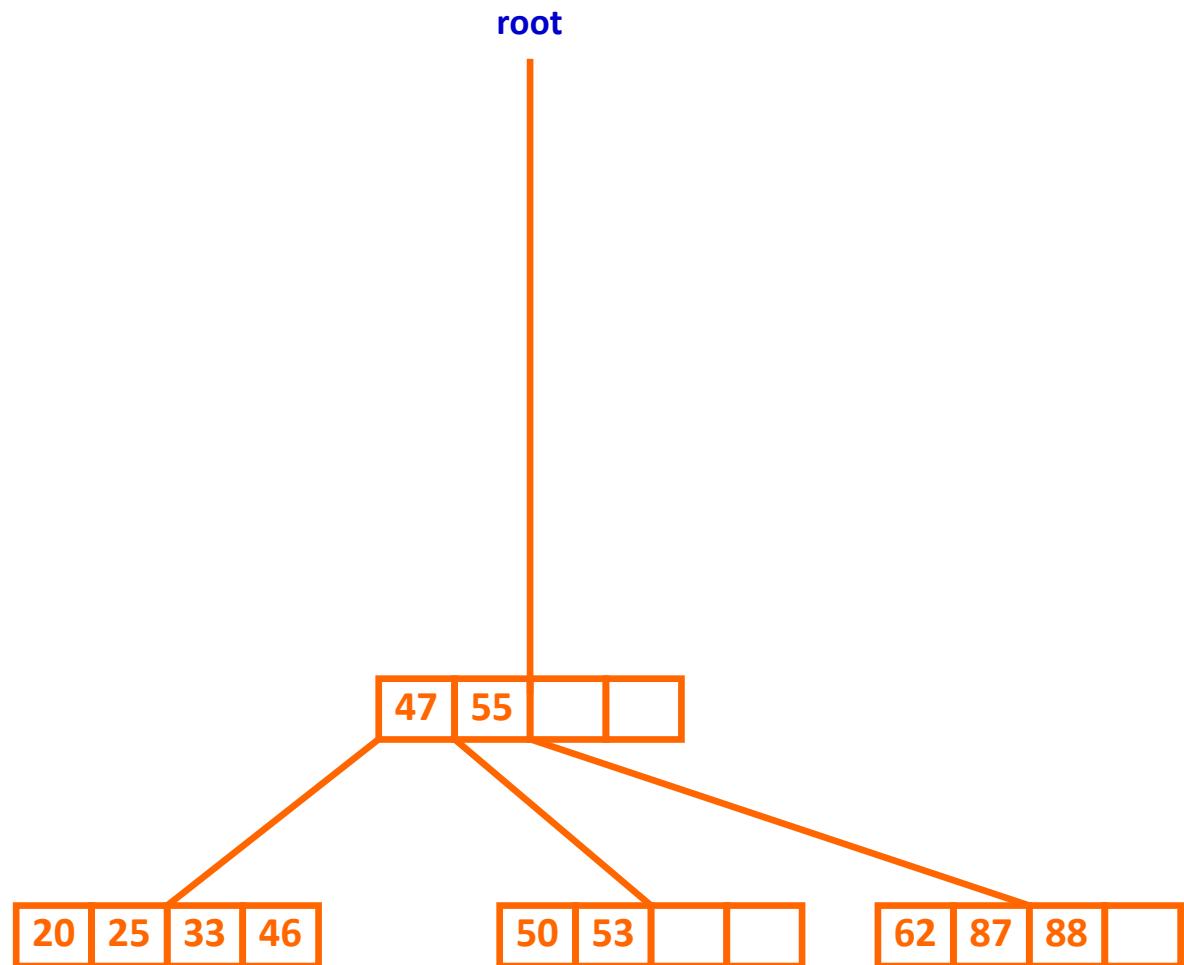
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)

5-way B Tree Insert Animation



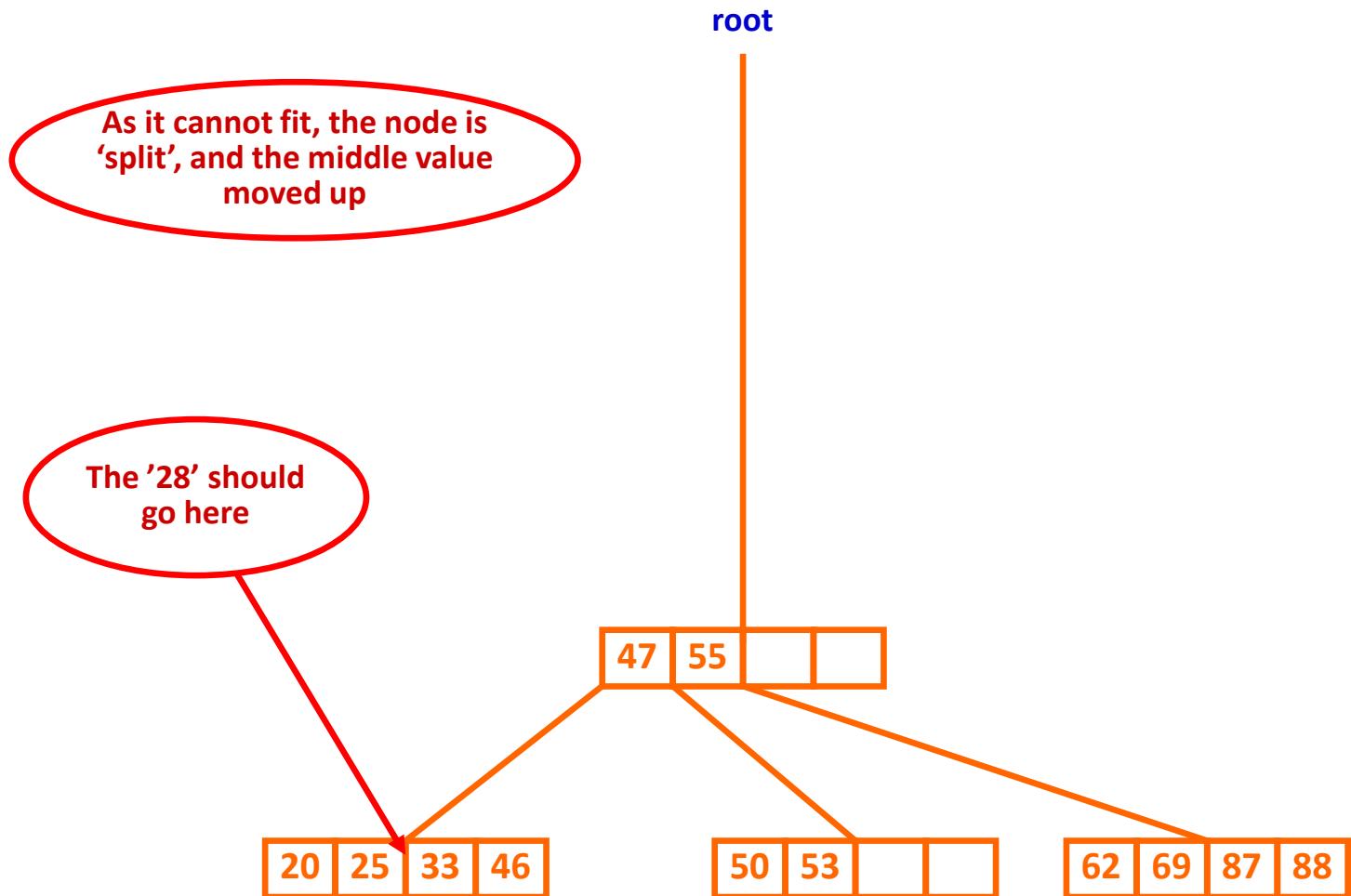
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)
Insert(69)

5-way B Tree Insert Animation



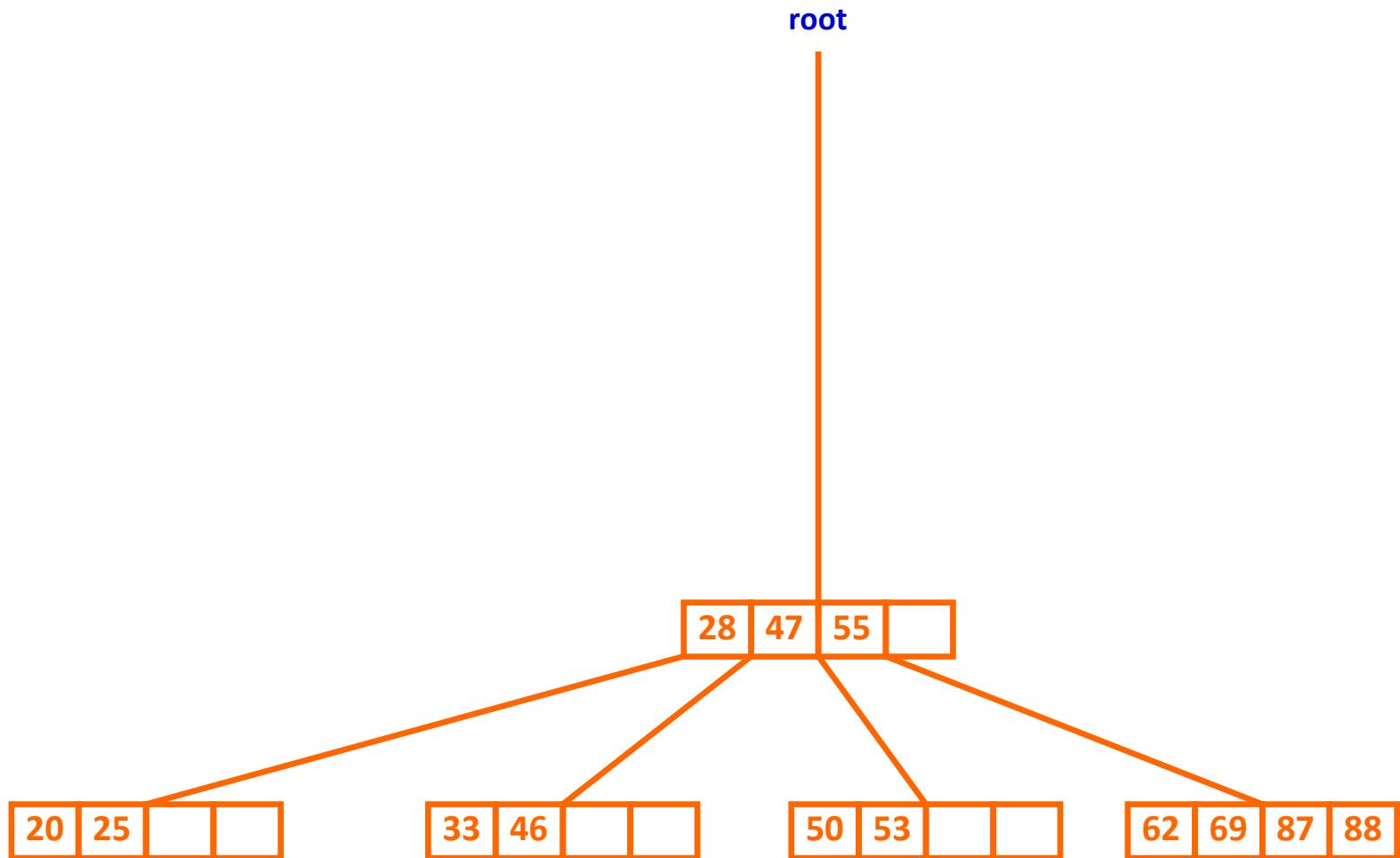
Insert(50)
Insert(20)
Insert(55)
Insert(47)
Insert(46)
Insert(33)
Insert(25)
Insert(87)
Insert(88)
Insert(53)
Insert(62)
Insert(69)
Insert(28)

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)

5-way B Tree Insert Animation



Multiway Tree vs B Tree

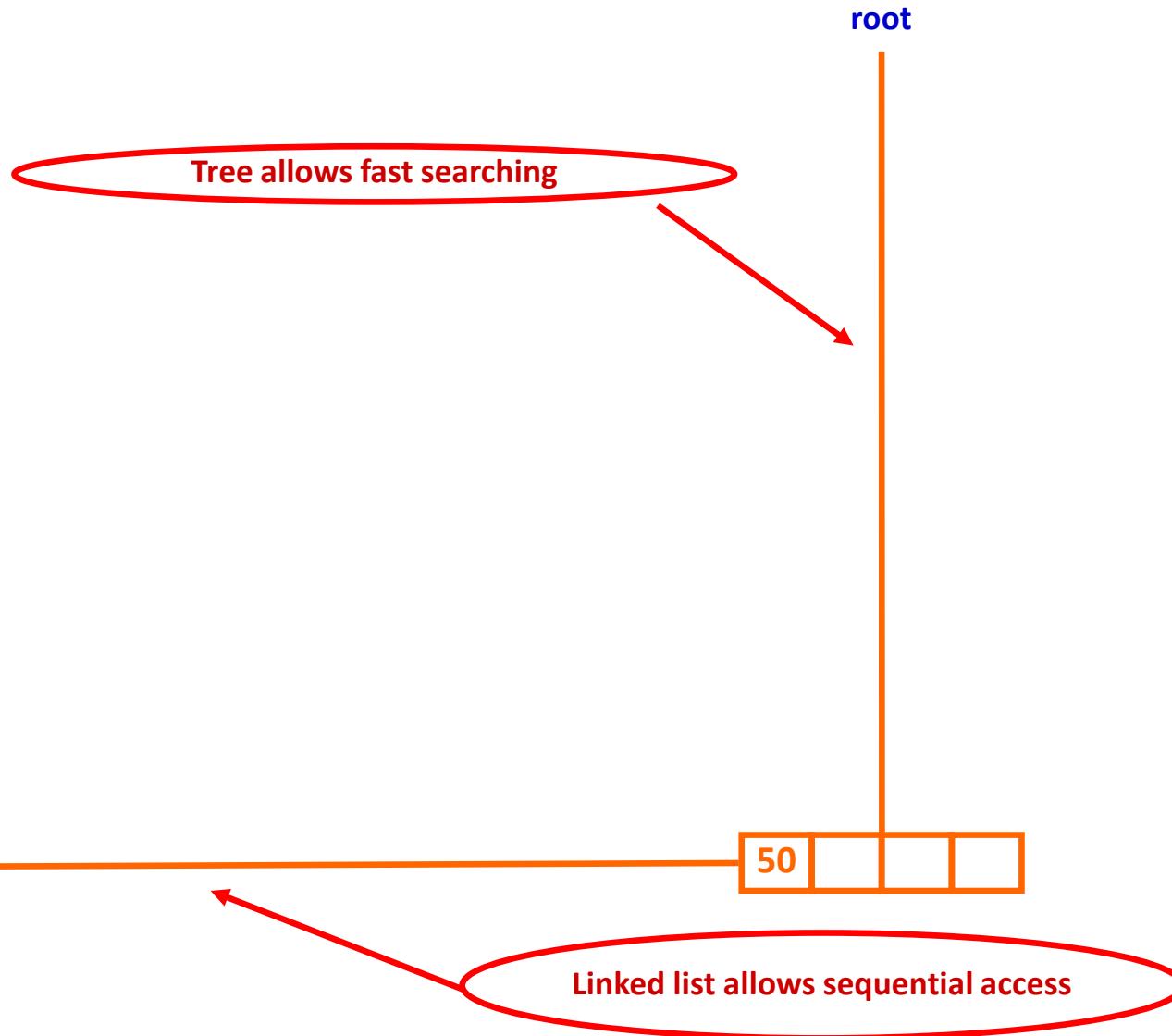
- The B Tree is clearly more efficient in terms of space.
- The B Tree is balanced and therefore has a lower search time.
- The B Tree, however, is more complicated to code.
- If **search time is important** (as with a database or list of objects in the scene of a game) the use of B Trees is essential.

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.

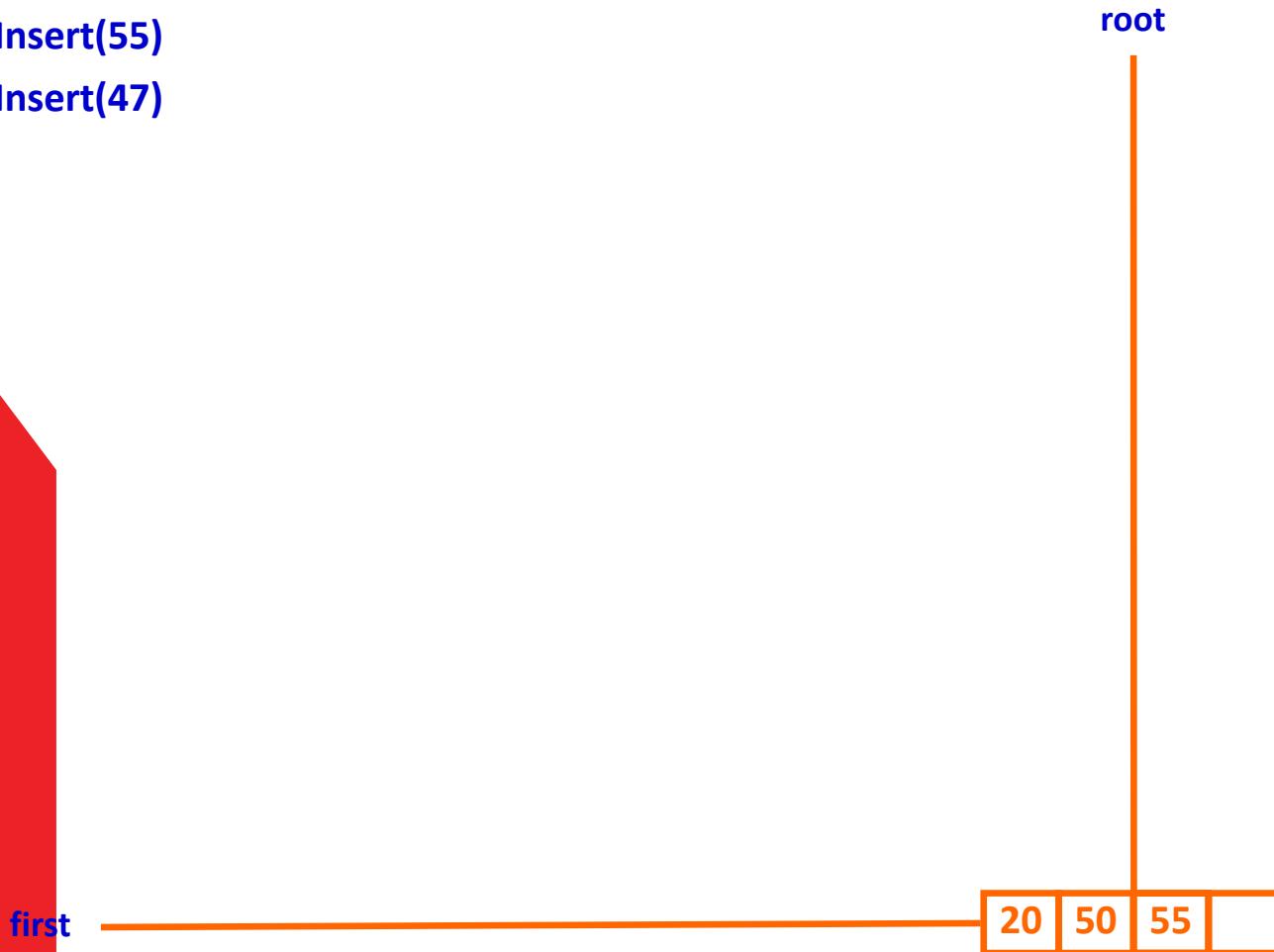
Insert(50)
Insert(20)

5-way B+ Tree Insert Animation



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

root

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

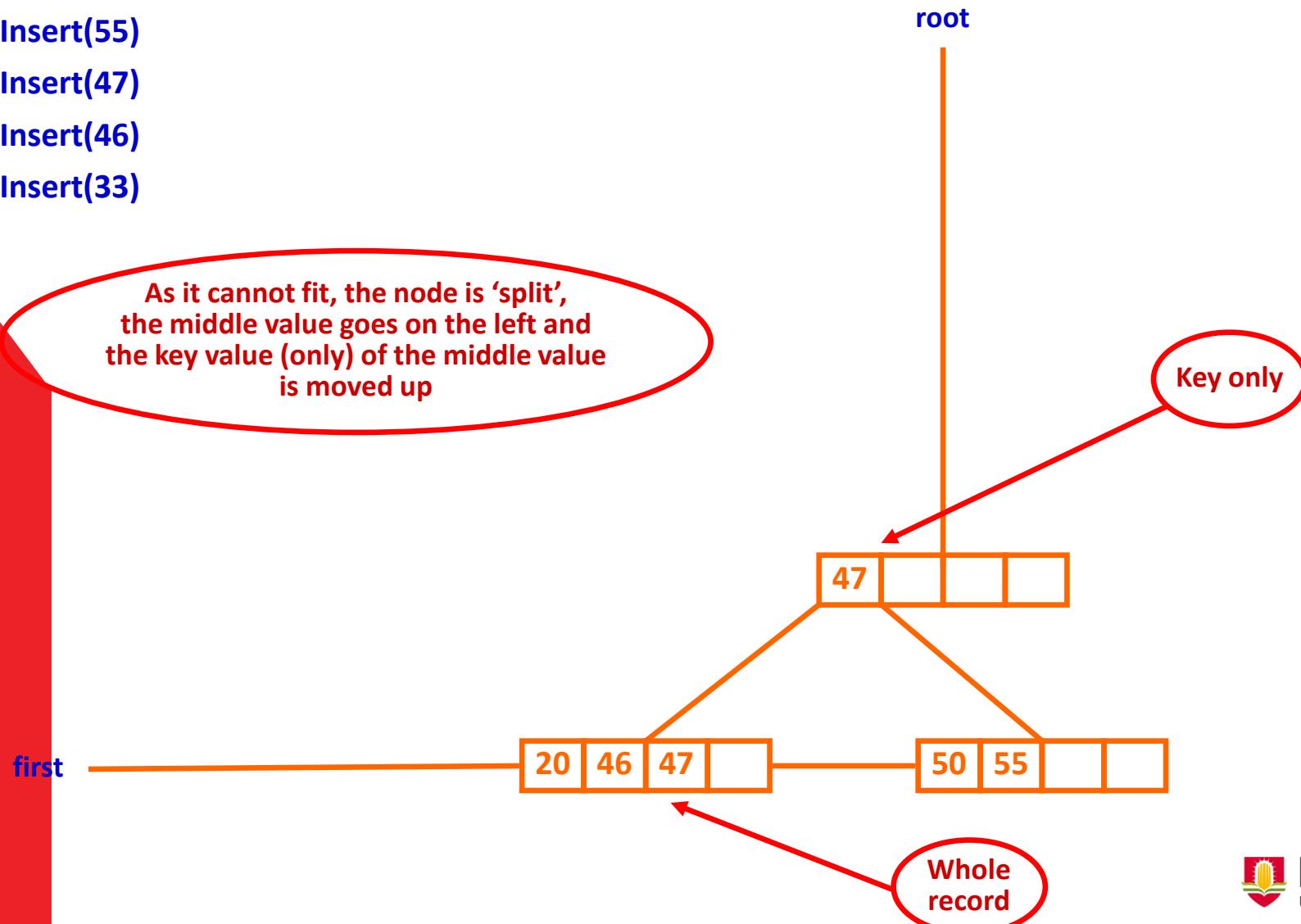
The '46' should
go here

first



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

The '25' should
go here



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



root

Keys
only

Whole records

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.

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](#).



Murdoch
UNIVERSITY

Data Structures and Abstractions

Hash Tables

Lecture 31



Important Advice for LAB/Assignment

- You must complete Lab 10.
 - Submission needed for the last assessed lab
 - Submission needed for the assignment
- Your BST needs to be usable beyond the purposes of the lab/assignment.
- Follow the assignment specifications carefully. Read the QandA file (when available) regularly to see any clarification or advice.
 - If the answer to your question is not there, ask early.
- Be mindful of summing small floating point numbers. Errors accumulate.
 - See the following for some advice:
 - https://en.wikipedia.org/wiki/Kahan_summation_algorithm
 - For a more detailed answer see: “What Every Computer Scientist Should Know About Floating-Point Arithmetic” at http://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Maps

- Previously we have looked at the STL map class, where any type can be used as a key into a container of paired values, giving direct access to the second part of the data.
- So we can have:

```
map<string, string> DictionaryType;  
  
DictionaryType dictionary; // not really a good way to name  
  
dictionary["aardvark"] = "A nocturnal mammal of southern  
Africa"  
  
cout << "aardvark:" << dictionary["aardvark"];  
  
//Work out how many string object constructions occurred in the lines  
//above
```

Hash Tables

- One way to achieve this kind of direct access for the map class is to use what is known as a *hash table*.
 - If keys are unique a balanced binary search tree can be used.
 - If keys are not unique and key are unordered as in `std::unordered_multimap` or `std::unordered_multiset`, then hashing is used.
- When storing the data, the key—in this case “aardvark” —is passed through a *hash function*, to give an index into an ordinary array. [1] [2]
- The quality of the hash function determines how many different keys hash to the same index value. (technically known as “collisions”)
- No hash function is perfect under all conditions, therefore there will always be clashes (“collisions”).
- Therefore there must also be a *collision resolution* defined.
- Hash tables will always have empty space. To work most efficiently they are generally required to be no more than half full.

Dealing with Strings

- The key used in the above example is a string.
- Obviously you cannot pass a string through a mathematical function.
- Therefore strings must be mapped to integers before hashing.
- There are many ways to do this, however it is important to make sure that the method chosen does not promote collisions.

Insertion into a Hash Table

- Insert (pair)
- index = HashFunction (pair.key)
index = index mod arraySize // in hash table i.e %

IF array[index] is empty
 array[index] = pair
ELSE
 HandleCollision (index, pair)
ENDIF
- End Insert

Searching a Hash Table

Search (key, found)

 index = HashFunction (key)

 index = index mod arraySize

 IF array[index] is empty

 found = false

 ELSE

 IF array[index].key = key

 found = true

 ELSE // another key was hashed to the same index

 found = CollisionSearch (index, pair)

 ENDIF

ENDIF

End Search

Hash Functions

- The ideal function would: [1]
 - be easy to calculate;
 - never produce the same index from two different keys;
 - spread the records evenly throughout the array;
 - deal with ‘bad’ keys better than others.
- Of course no function has all of these attributes under all conditions.
 - It may be possible under restricted conditions where all keys are known in advance.
- Common Hash Functions
 - Truncation
 - Extraction
 - Folding
 - Modular arithmetic
 - Prime number division
 - Mid-square hashing
 - Radix conversion

Radix Conversion

- The best known of these is Radix Conversion.
- Choose a low prime [1] number such as 7, 11 or 13 to use as the base of a polynomial. Then use the digits of the key as the factors of the polynomial.
- Finally modulate by the array size, which should be a prime number.
- For example, if

key = 32934648

array size = 997

Base = 7

Then

$$\begin{aligned}\text{index} &= (3 * 7^7 + 2 * 7^6 + 9 * 7^5 + 3 * 7^4 + 4 * 7^3 + 6 * 7^2 + 4 * 7 + 8) \text{ MOD } 997 \\ &= 2866095 \text{ MOD } 997 \\ &= 717\end{aligned}$$

Collision Resolution

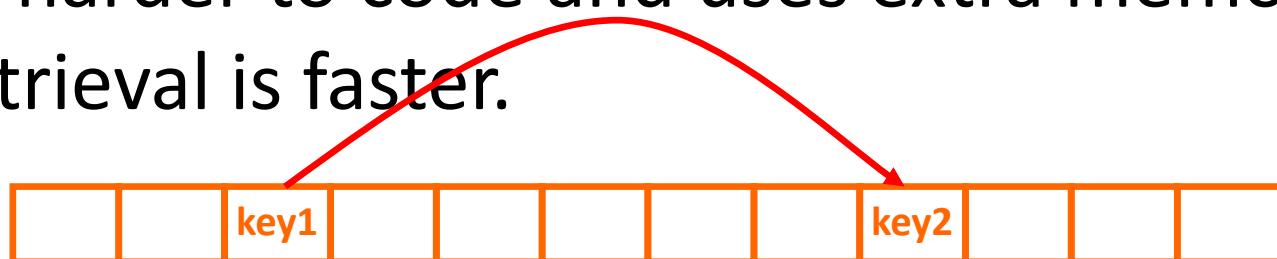
- Collision resolution needs to:
 - avoid clustering of records;
 - be as simple to code as possible;
 - only fail when the array is actually full;
 - be ‘reversible’ to allow for deletion/search.
- As before, no method fulfils all these requirements under every possible condition.
- Common collision resolutions are:
 - Linear probing
 - Quadratic probing
 - Random probing
 - Linked collisions
 - Overflow containers

Probing

- **Linear** probing simply looks for the **next empty space** in the array. So if index is full, index+1 is checked, then index+2, index+3 etc. This has the disadvantage of increasing clustering and therefore collisions.
- **Quadratic** probing looks for the next empty space using ‘square’ jumps. So if index is full, index+1 is checked, then index+4, index+9, index+16 etc. This avoids the clustering of linear probing, but can fail when the array is not full.
- **Random** probing uses a random number generator—from a set starting point—for the increments in index. This avoids clustering, but is harder to reverse when a record is to be deleted (search?). Use pseudo-random number generator.

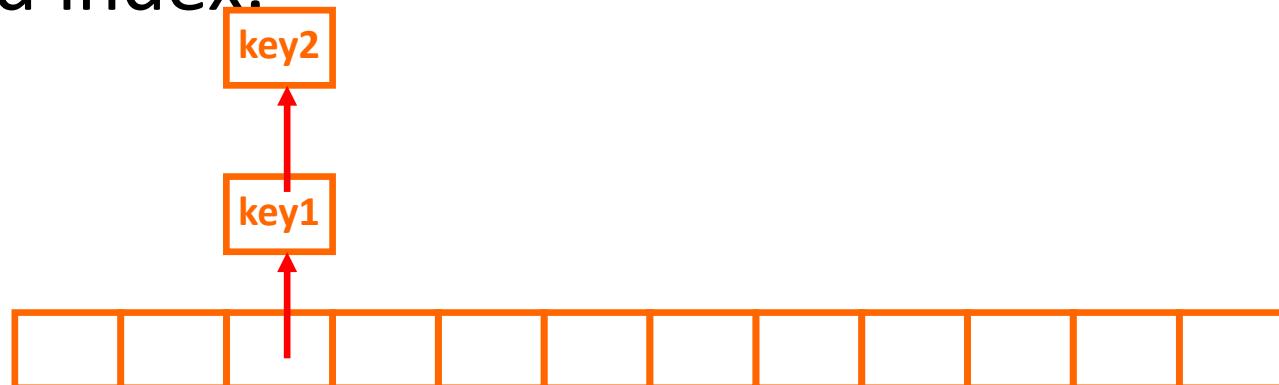
Linking Collisions

- After the first collision, a second hash function is used to generate an alternate position and the two are linked.
- A third collision would then have a link from the second collision and so on.
- This is harder to code and uses extra memory, but retrieval is faster.



An Overflow Container

- Instead of using a one dimensional array to store data, a two dimensional structure is used.
- The records are placed in a linked list from the hashed index.



Readings

- Reference book, Introduction to Algorithms.
Chapter on Hash Tables.

Further Exploration

- Khan Academy Video one particular example of the use Hash functions [“Bitcoin: Cryptographic hash functions”](#)
- Tutorial on Hash functions
<http://research.cs.vt.edu/AVresearch/hashing/>



Murdoch
UNIVERSITY

Data Structures and Abstractions

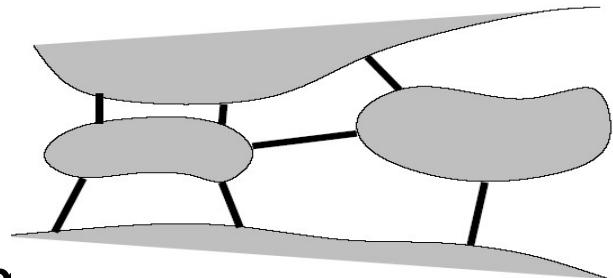
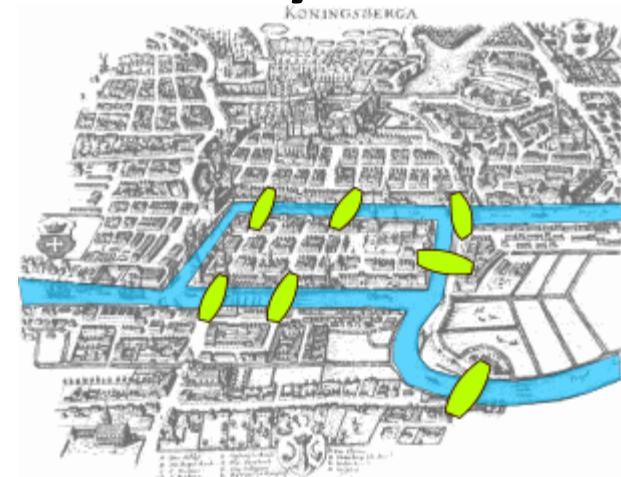
Graph Theory

Lecture 32



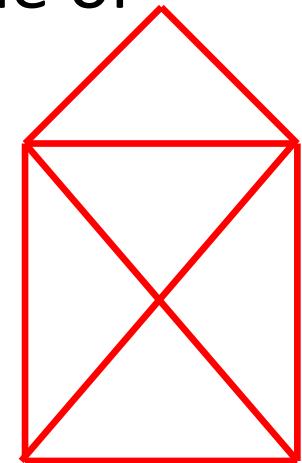
The Origins of Graph Theory

- Graph Theory (unlike a lot of what we do) dates back to before 1736.
- In Konisberg there were two islands in the middle of a river, connected by 7 bridges. [1]
 - Textbook has the abstract version.
- The question was: “is it possible to cross each bridge exactly once?”
 - Abstract representation is used to investigate solutions.
 - Any solution obtained can then be used for similar problems.
- In 1736, Euler answered this problem, by establishing “Graph Theory” as a discipline. (The answer is “no”)



Another Common Problem

- As a child you may have met something similar:
Draw the shape below without taking your pen off the page and without going over any line or node more than once.
- It is a graph problem, just as the Konisberg problem is a graph problem.

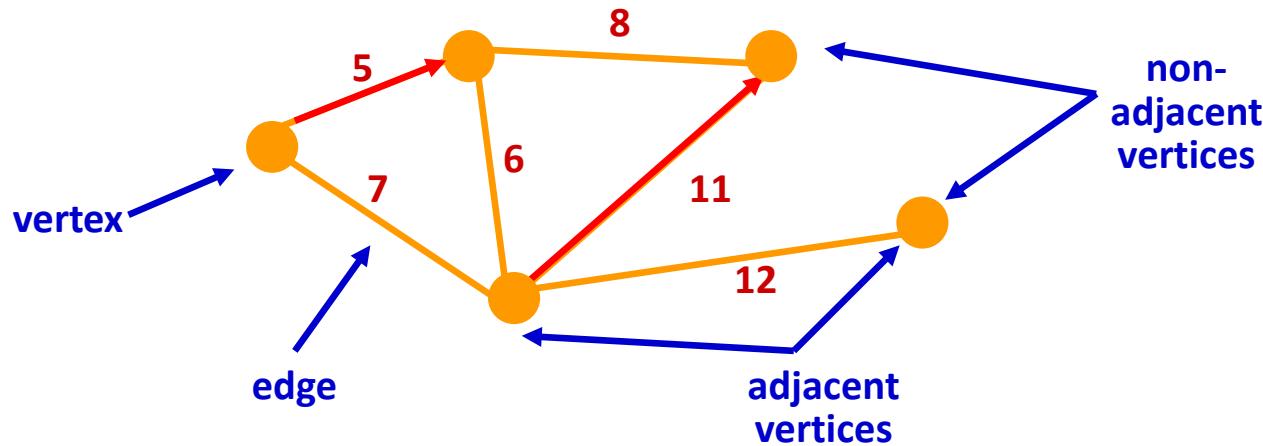


But...

- But the theory itself remained a kind of mathematician-only esoteric field until
 1. Computers became available that could handle graph processing algorithms in reasonable time.
 2. Many of the complex problems of society were recognised to be graph problems.
 3. It was realised that Network traffic and the WWW were graphs.
 4. Some AI applications (simulations, neural networks etc) were discovered to use graph theory.
 5. Computer game playing required graph theory.

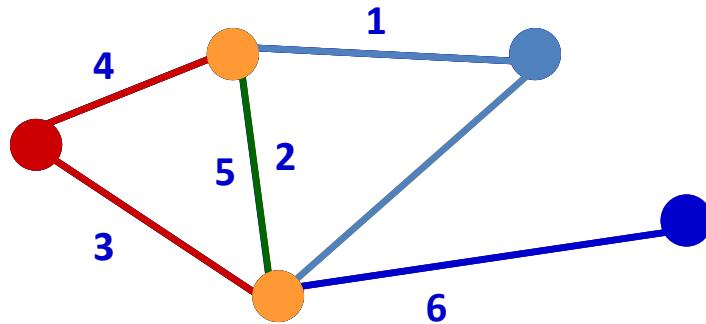
So What is a Graph?

- A graph is a set of *vertices* connected by *edges*.



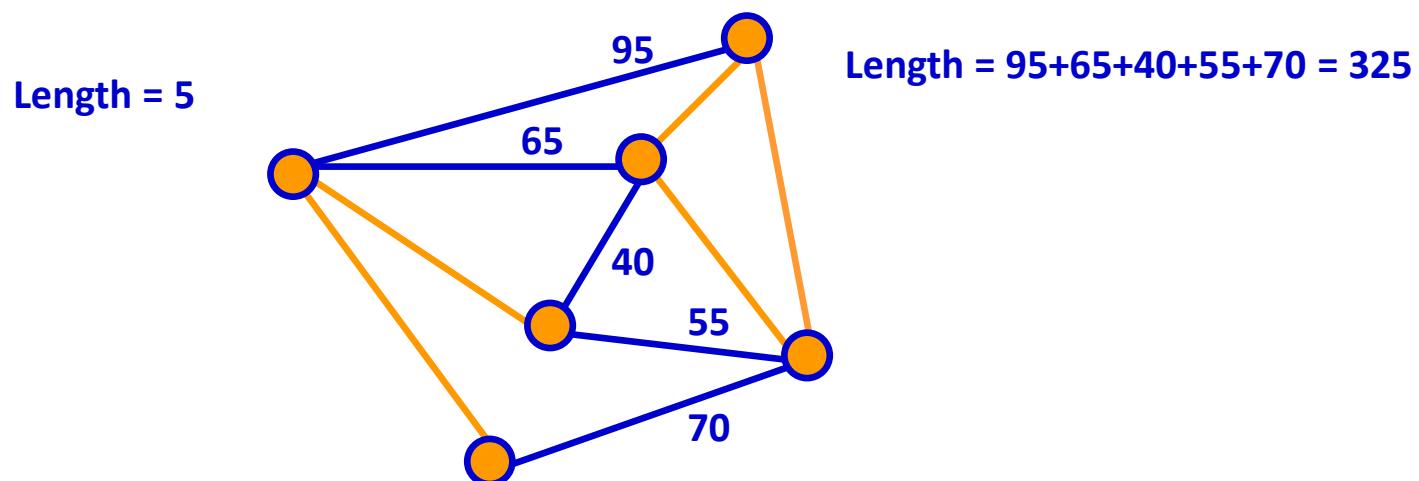
- Two vertices are *adjacent* if they are connected by a single edge.
- A graph is *weighted* if there is a number associated with each edge. (can be cost, distance, ..etc)
- A graph is *directed* if any of the edges are one-way.

Graph Definitions

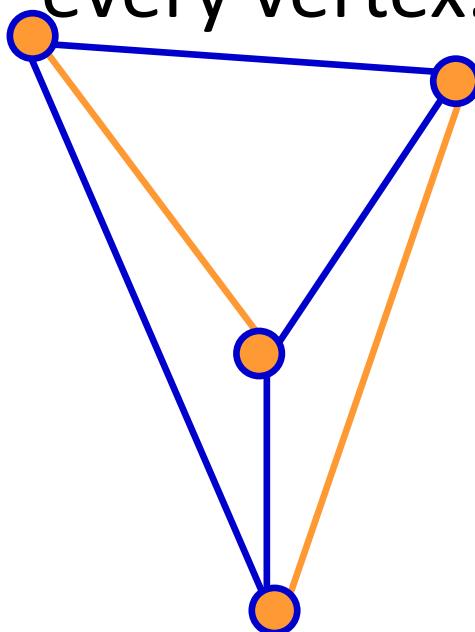


- A *path* is a sequence of adjacent vertices
- A *simple path* is one that has distinct edges: no vertex is visited twice.
- A *cyclic path* is one where the start and finish are the same vertex.
- Two paths are *disjoint* if they have no vertices in common, other than, possibly, their endpoints. [see the red and blue paths]

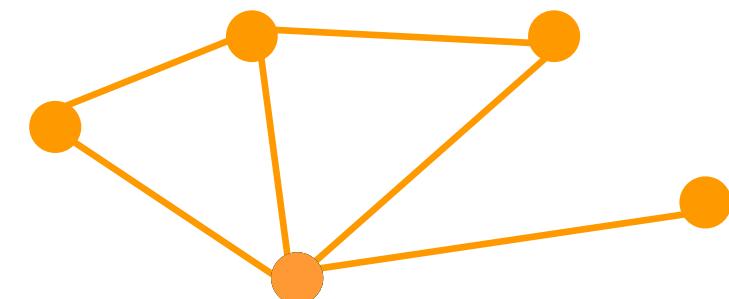
- In an unweighted graph, the length of a path is the number of traversed edges.
- In a weighted graph, the length of a path is the sum of the weights of the traversed edges.



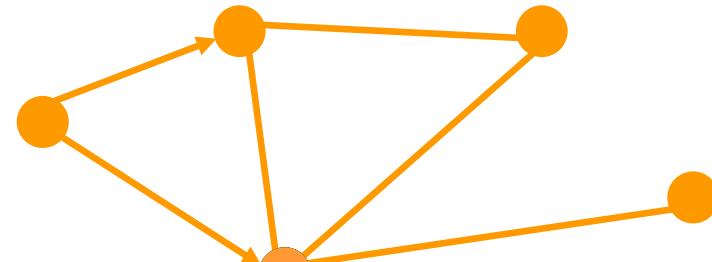
- A *tour* is a cyclic path that touches every vertex.



- A connected graph is one where every vertex is reachable from every other vertex

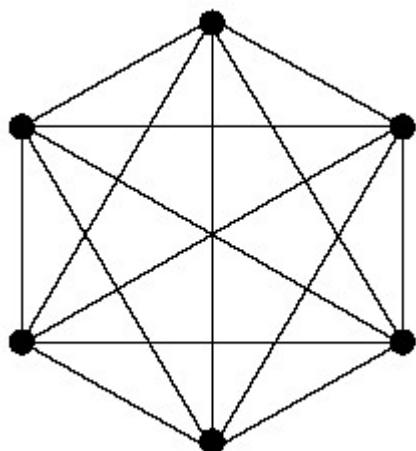


Connected

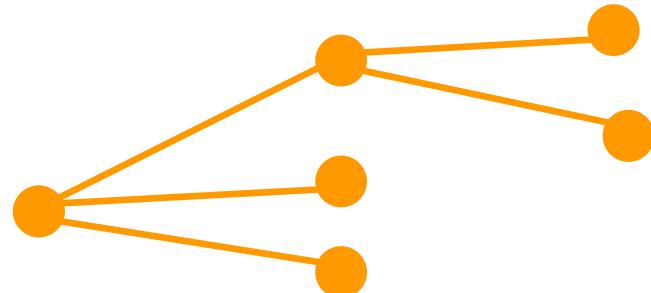


Not connected

- A *complete* graph is one where every vertex is adjacent to every other vertex.



- A graph with no cycles (an *acyclic graph*) is a tree.

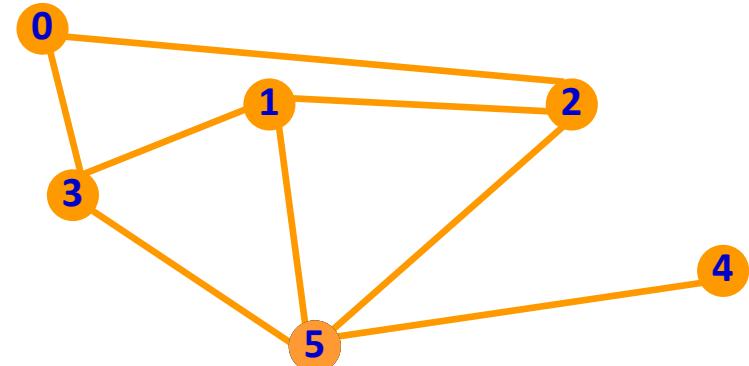


Data Structures to Represent Graphs

- Representing a graph as vertices and edges is fine in the abstract (physical) sense but makes processing too difficult.
- Two alternatives are therefore used within programming:
 - Adjacency matrices
 - Constant access time
 - Slow search time
 - Adjacency list
 - Fast search time
 - Slow access time
- For both of these, the vertices are arbitrarily numbered.

Adjacency Matrix Representation

- The graph is represented as a two dimensional array of boolean.
- A vertex is not considered to connect to itself.

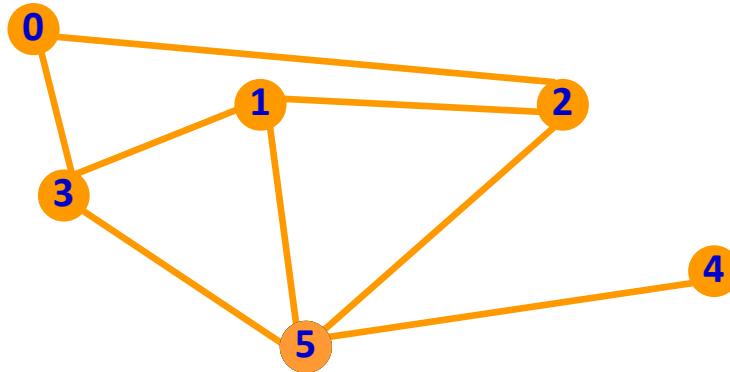


	0	1	2	3	4	5
0	false	false	true	true	false	false
1	false	false	true	true	false	true
2	true	true	false	false	false	true
3	true	true	false	false	false	true
4	false	false	false	false	false	true
5	false	true	true	true	true	false

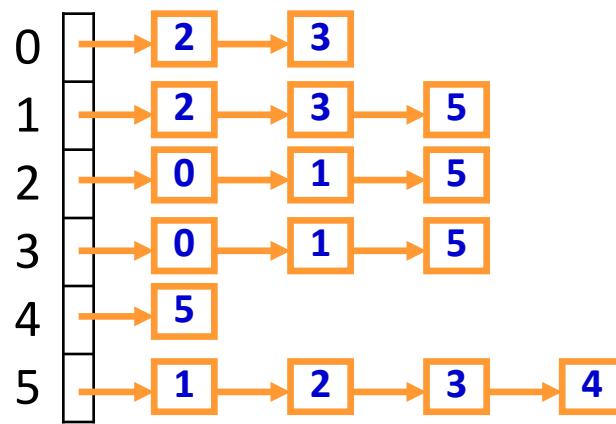
Drawing an Adjacency Matrix

- Make sure you can draw an adjacency matrix for a graph. Use the Graph program to check your answers.

Adjacency List Representation



- The graph is represented as a one dimensional sorted list of connected vertices:



Drawing an Adjacency List

- Make sure you can draw an adjacency list for a graph. Use the Graph program to check your answers.

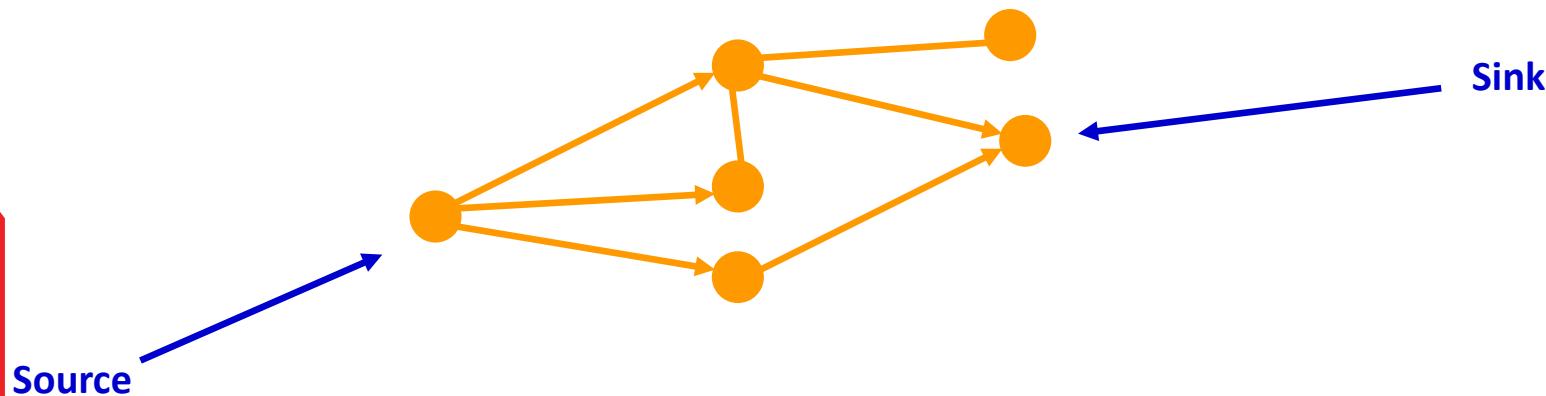
Matrix and List Comparison

- Advantages of Lists
 - More flexible as the size is not fixed
 - Less space used: $O(V+E)$ rather than $O(V^2)$ for a matrix.
 - Faster processing (searching) at each vertex
- Advantages of Matrices
 - Easier to program
 - Access time to find out if a pair of vertices are connected is constant time as opposed to $O(V)$ for lists.

Directed Graph Definitions

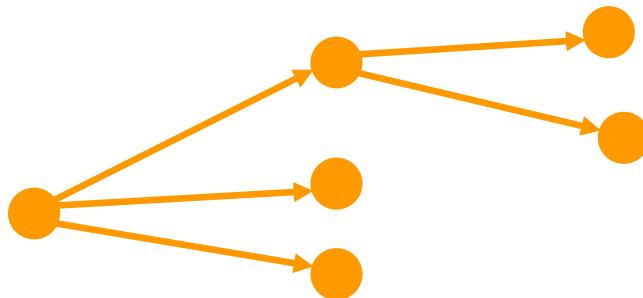
- Directed graphs are also known as di-graphs.
- A vertex is *reachable* from another vertex if there is a path between them.
- It is assumed that each vertex can reach itself.
- The *in-degree* of a vertex is the number of edges leading into a vertex.
- The *out-degree* of a vertex is the number of edges leading out of a vertex.

- A *sink* is a vertex with out-degree of zero.



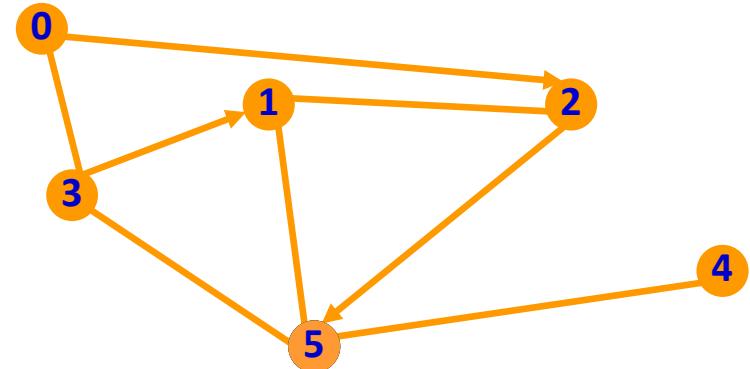
- A *source* is a vertex of in-degree 1: it is reachable only from itself.

- A *map* is a di-graph where every vertex has out-degree 1.
- A di-graph is *strongly connected* if every vertex is reachable from every other vertex.
- A di-graph with no cycles is an Acyclic Directed Graph, or DAG.



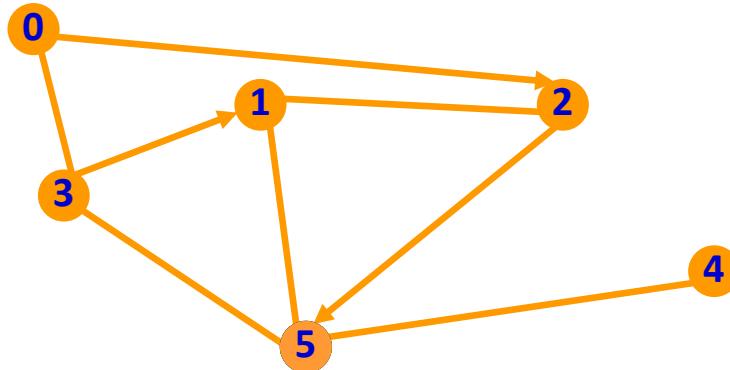
Adjacency Matrix Representation of a Di-graph

- The di-graph is represented as a two dimensional array of boolean.
- Note that in a di-graph vertices are considered to be connected to themselves.

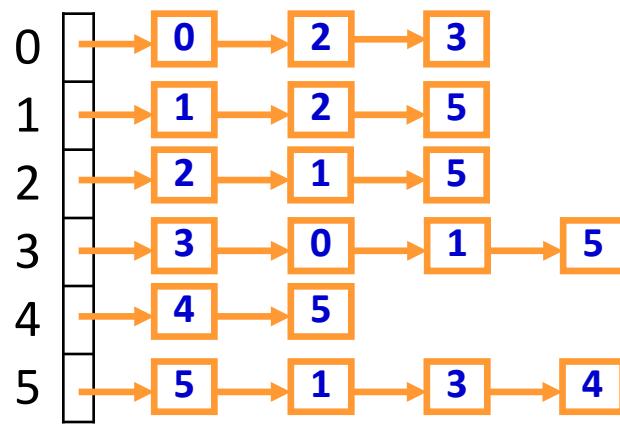


	0	1	2	3	4	5
0	true	false	true	true	false	false
1	false	true	true	false	false	true
2	false	true	true	false	false	true
3	true	false	false	true	false	true
4	false	false	false	false	true	true
5	false	true	false	true	true	true

Adjacency List Representation



- The di-graph is represented as a one dimensional sorted list of connected vertices:



Graph Domains

- Social media – friendship networks
 - Interconnections in ecosystems
 - Genetics and ancestry
 - Chemical structures
 - Traversal problems
 - Travel itineraries
 - Neural networks
 - The WWW (the biggest graph of them all)
 - Electric circuits
 - Scheduling
 - Financial transactions
 - Compilers use graphs to represent call structures
 - Within games software
 - UML diagrams, data flow diagrams, E-R diagrams etc
 - Automatic diagram generation
- etc.

Some Graph, Di-Graph and DAG Processing Problems

- Searching: how do we get from a particular vertex to another.
- Connectivity: is a given graph connected.
- Find the minimum length set of edges that connects all vertices (the Minimum Spanning Tree or MST).
- Find the shortest path between two vertices.
- Find the shortest path from a specific vertex to all other vertices (the Shortest Path Tree or SPT).
- Planarity: can a specific graph be drawn without any intersecting lines?
- Matching: what is the largest subset of edges with the property that no two are connected to the same vertex?
- Find the tour with the shortest path (mail carrier problem).
- Topological Sort: sort the vertices of a DAG in order of the number of dependencies.

Complex problems

- Graphs are a powerful tool for modelling complex problems.
- *“The great unexplored frontier is complexity – I am convinced that nations and people that master the new science of complexity will become the economic, cultural, and political superpowers of the next century.” Heinz Pagels*

In Fact

- These problems are NP-Hard.
- There is no solution for any of them that is guaranteed to be solvable in a reasonable amount of time.
 - Restricted case solutions are possible but not in the general case.
- There are only solutions that work quite well in some circumstances.
- This, combined with the large number of domains, makes this field one that is rich in research possibilities.

Readings

- Textbook Chapter on Graphs.
- The lecture notes and textbook is sufficient for this unit.
- Further exploration:
 - [Graphs and Networks](#) has interesting applications of graphs, including a wall chart.
 - How graph problems gave the [biggest finite number known that no one can](#) write but ends in 7.
 - [Interactive explanations of various algorithms \[1\]](#)
 - [Complex systems: Network thinking](#), Melanie Mitchell, Artificial Intelligence, vol. 170(18), Science Direct, Elsevier.
 - Reference book, Introduction to Algorithms. Part on Graph Algorithms contains a number of chapters on graph algorithms. (for further study)
 - Consider doing the unit MAS225 where Graphs are covered in detail. <https://handbook.murdoch.edu.au/units/12/MAS225>