University of Central Lancashire
UCLan

CO2412 Computational Thinking

Arrays and linked lists: Overview
Sorting: Overview
Tree data structures: Introduction

Where opportunity creates success

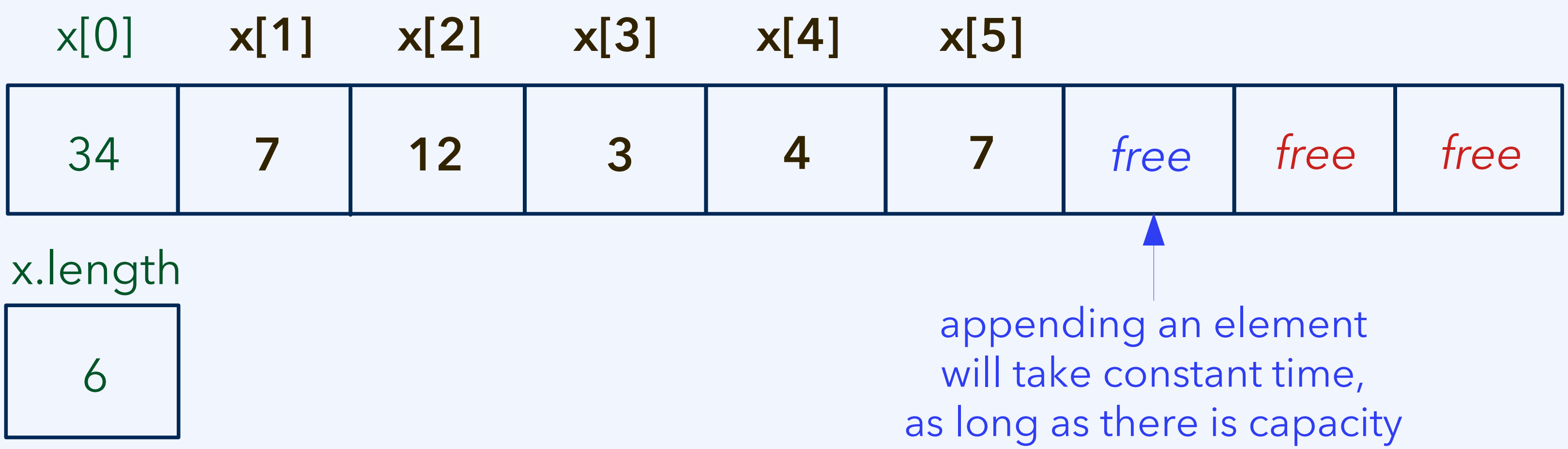
Arrays and linked lists: Overview

CO2412

23rd November 2021

Dynamic arrays: Efficiency analysis

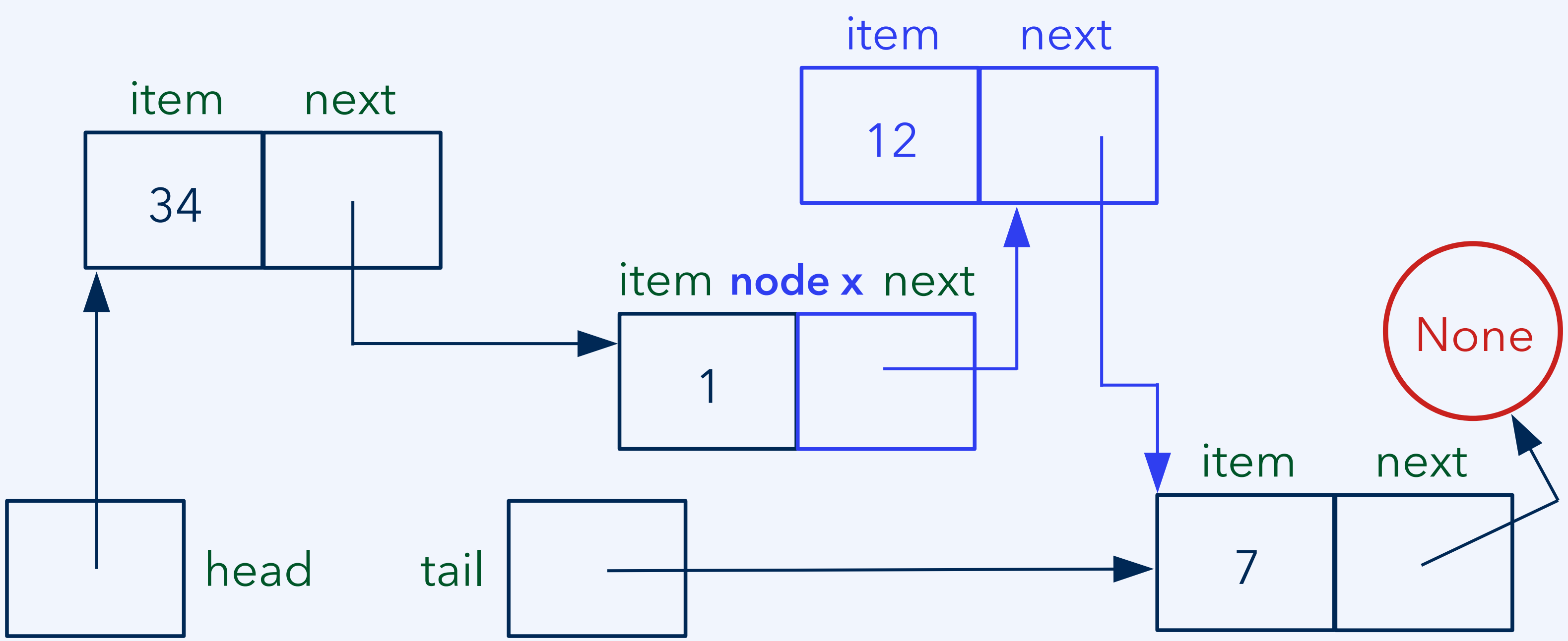
- Read/write access to an array element: O(1) time.
  Address of the i-th element computable by pointer arithmetics.
- Deleting an element from the array: **O(1) at the end, O(*n*) elsewhere.**
  All the elements with greater indices need to be shifted.
- **Extending the array by one element?** O(1) at the end, if there is capacity.
  O(*n*) elsewhere, or if the capacity of the dynamic array is exhausted.



Singly linked lists

If a **reference to a node** is given, another item can be **inserted after** that node in constant time; by accessing the linked-list object, it takes constant time to insert a new head at the beginning (**push**) or a new tail at the end (**append**).

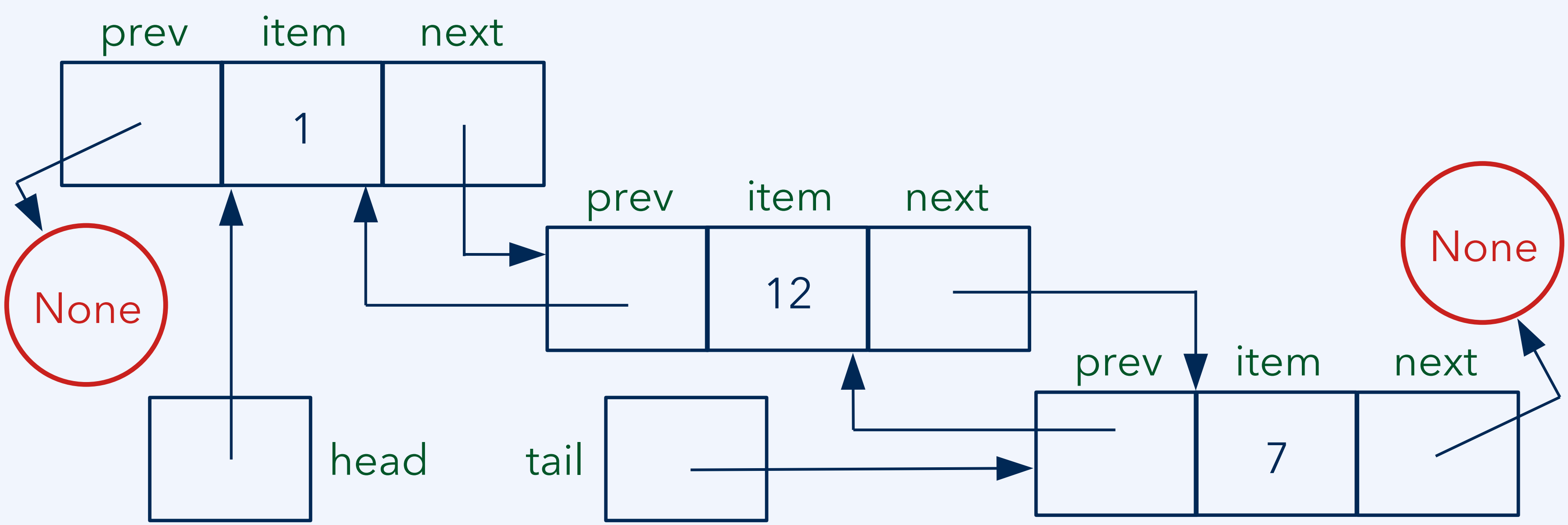
Example: Insert 12 after node x, to which we already have a reference.



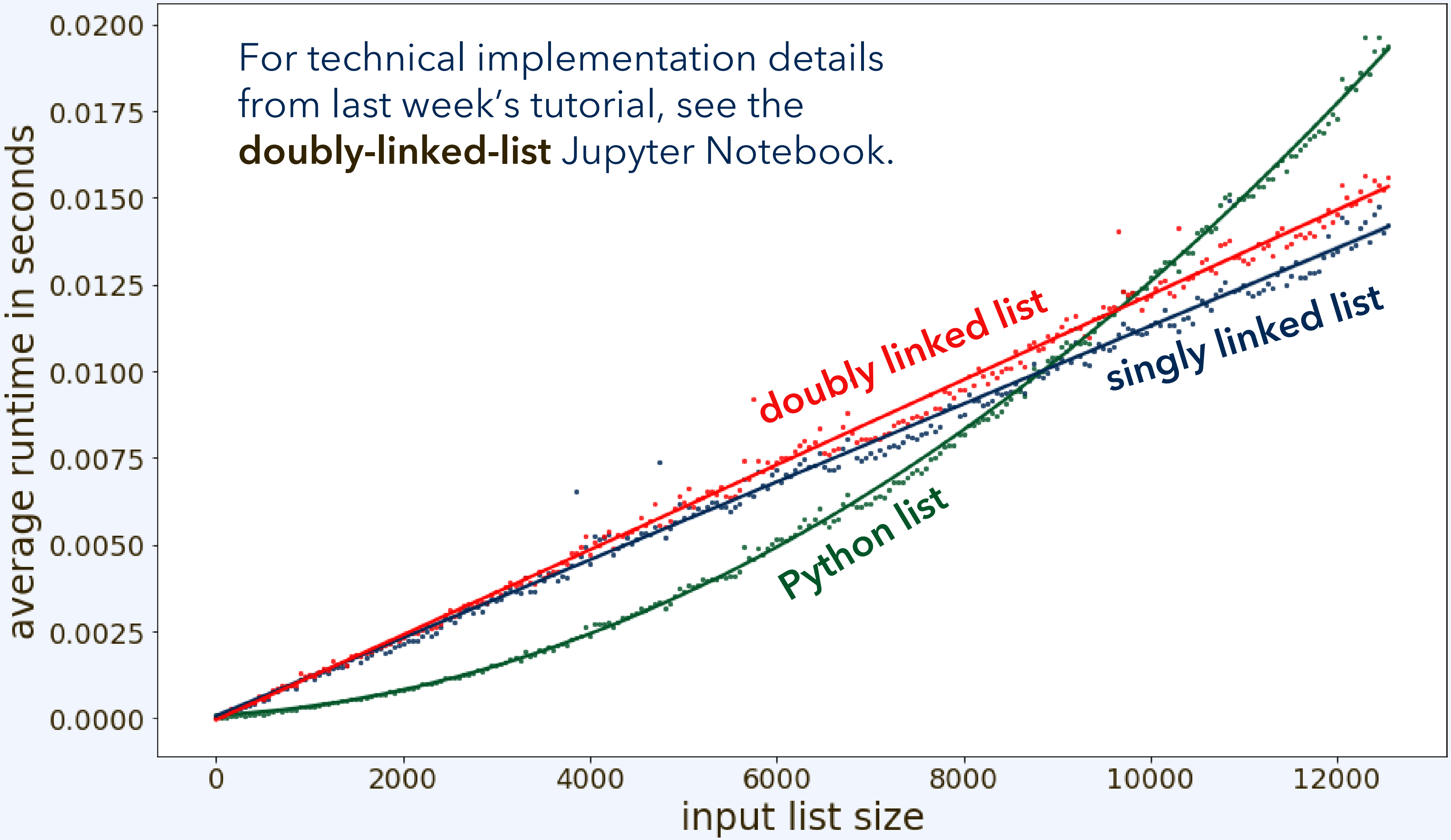
Doubly linked lists

In a **doubly linked list**, each node also contains a reference (or pointer) to the **previous node**. This facilitates traversal in **both directions and inserting** a new data item **before** any given node (rather than only after it), all in constant time.

Singly linked lists require two variables per data item (item and next).
Doubly linked lists require three variables per data item (prev, item, and next).



Mod-3 copying performance comparison



Summary: Efficiency analysis

- Read/write access to a data item at position *k*
  - For a dynamic array, O(1) time; fast access by pointer arithmetics
  - For a singly linked list, O(k) time, *i.e.*, O(n) in the average/worst case

- Iterating over the data, *i.e.*, proceeding from one item to the next one
  - O(1) both for dynamic arrays and for linked lists;
    in a *singly* linked list, this is limited to one direction, forward

- Deleting a data item at position *k*
  - For a dynamic array, O(1) at the end, O(n – k) in general
  - For a singly linked list, O(1) at the head, or if we have a reference to the element at position k–1; otherwise, in general, O(k)

Above, n is the length (logical size) of the dynamic array or linked list.

Remark: For **dynamic arrays**, *lookup* and jumping to an index are in O(1). It is only *rearranging the elements in memory* that, in general, can take linear time.

Summary: Efficiency analysis

- Read/write access to a data item at position k
  - For a dynamic array, $O(1)$ time; fast access by pointer arithmetics
  - For a singly linked list, $O(k)$ time, *i.e.*, $O(n)$ in the average/worst case
  - For a doubly linked list, $O(\min(k, n - k))$, which is still effectively $O(n)$

- Iterating over the data, *i.e.*, proceeding from one item to the next one
  - $O(1)$ both for dynamic arrays and for linked lists

- Deleting a data item at position k
  - For a dynamic array, $O(1)$ at the end, $O(n - k)$ in general
  - For a singly linked list, $O(1)$ at the head, or if we have a reference to the element at position k–1; otherwise, in general, $O(k)$
  - For a doubly linked list, $O(1)$ at the head or tail, or if we have a reference to that region of the list; in general, $O(\min(k, n - k))$

Remark: For **linked lists**, *insertion/deletion as such* takes constant time, once the node has been localized. However, *getting to the node* can take $O(n)$ time.

Summary: Efficiency analysis

- Read/write access to a data item at position k
  - For a dynamic array, O(1) time; fast access by pointer arithmetics
  - For a singly linked list, O(k) time, *i.e.*, O(n) in the average/worst case
  - For a doubly linked list, O(min(k, n – k)), which is still effectively O(n)

- Iterating over the data, *i.e.*, proceeding from one item to the next one
  - O(1) both for dynamic arrays and for linked lists

- Inserting an additional data item at position k
  - For a dynamic array, O(n) in the worst case, *i.e.*, whenever the capacity is exhausted; with free capacity, O(1) at the end, O(n – k) elsewhere
  - For a singly linked list, O(1) at the head or tail, or if we have a reference to the element at position k–1; Otherwise, in general, O(k)
  - For a doubly linked list, O(1) at the head or tail, or if we have a reference to that region of the list; in general, O(min(k, n – k))

Application: Stacks and queues

- **Stacks** function by the principle **“last in, first out” (LIFO)**
  - Can be implemented using a singly linked list:
    - » Attach (push) new elements at the head of the list only
    - » Detach (pop) elements from the head of the list only
  - Can be implemented using a dynamic array:
    - » Attach (push) new elements at the end of the array only
    - » Detach (pop) elements from the end of the array only
- **Queues** function by the principle **“first in, first out” (FIFO)**
  - Can be implemented using a singly linked list (with a tail reference):
    - » Attach (push) new elements at the tail of the list only
    - » Detach (pop) elements from the head of the list only

All these operations can be carried out in constant time;
in case of the push operation for the dynamic array, subject to free capacity.

Sorting: Overview

CO2412

23rd November 2021

Sorting: Overview

For sorting, we have so far compared:

- **Selection sort**, a greedy algorithm with **O(n^2)** time efficiency
- **Mergesort**, a divide-and-conquer algorithm with **O(n log n)** efficiency

By a statistical argument it can be proven that O(n log n) is the best theoretically possible efficiency of a sorting algorithm, i.e., it the **time complexity of the sorting problem** is O(n log n).

Sorting: Overview

For sorting, we have so far compared:

- **Selection sort**, a greedy algorithm with **O(n^2)** time efficiency
- **Mergesort**, a divide-and-conquer algorithm with **O(n log n)** efficiency

By a statistical argument it can be proven that O(n log n) is the best theoretically possible efficiency of a sorting algorithm, i.e., it the **time complexity of the sorting problem** is O(n log n).

Rough summary of the argument:

- For a list with n elements, there are $n \cdot n{-}1 \cdot \ldots = n!$ permutations, *i.e.*, possible ways in which the list may need to be rearranged.
- Which of these permutations is correct can only be determined by comparing elements.

Sorting: Overview

For sorting, we have so far compared:

- **Selection sort**, a greedy algorithm with **O(n^2)** time efficiency
- **Mergesort**, a divide-and-conquer algorithm with **O(n log n)** efficiency

By a statistical argument it can be proven that O(n log n) is the best theoretically possible efficiency of a sorting algorithm, i.e., it the **time complexity of the sorting problem** is O(n log n).

Rough summary of the argument:

- For a list with n elements, there are $n \cdot n\text{–}1 \cdot \ldots = n!$ permutations, *i.e.*, possible ways in which the list may need to be rearranged.
- Which of these permutations is correct can only be determined by comparing elements. With each comparison operation, which returns True or False, we can at best distinguish between two options.

Sorting: Overview

For sorting, we have so far compared:

- **Selection sort**, a greedy algorithm with **O(n^2)** time efficiency
- **Mergesort**, a divide-and-conquer algorithm with **O(n log n)** efficiency

Rough summary of the argument:

- For a list with n elements, there are $n \cdot n{-}1 \cdot \ldots = n!$ permutations, *i.e.*, possible ways in which the list may need to be rearranged.
- Which of these permutations is correct can only be determined by comparing elements. With each comparison operation, which returns True or False, we can at best distinguish between two options.

- Therefore, with k operations, we can at best distinguish 2^k options.
- If we have $n!$ options, we need at least $\log n!$ operations.
- However, $O(\log n!)$ is the same as $O(n \log n)$.

Insertion sort: Another sorting algorithm

For sorting, we have so far compared:

- **Selection sort**, a greedy algorithm with **O(n^2)** time efficiency
- **Mergesort**, a divide-and-conquer algorithm with **O(n log n)** efficiency

We could be satisfied with mergesort, which has the optimal asymptotic efficiency. However, many applications require maintaining a sorted list.

Insertion sort is a sorting algorithm that **keeps inserting into a sorted list**:

Test list: [35, 16, 58, 3, 11, 106, 15, 55, 7, 81, 1]

Step 1: [35] → Step 2: [16, 35] → Step 3: [16, 35, 58] → Step 4: [3, 16, 35, 58]

… → Step 11: [1, 3, 7, 11, 15, 16, 35, 55, 58, 81, 106]

Insertion sort

```
def insertion_sort(x):

    for i in range(len(x)):

        j = 0
        element_i = x[i]

        while x[j] < element_i and j < i:
            j += 1

        if i != j:
            x.pop(i)
            x.insert(j, element_i)
```

```
Test list: [41, 17, 71, 4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:1] is sorted;
Sorted part of the list: [41]
Unsorted part of the list: [17, 71, 4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:2] is sorted;
Sorted part of the list: [17, 41]
Unsorted part of the list: [71, 4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:3] is sorted;
Sorted part of the list: [17, 41, 71]
Unsorted part of the list: [4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:4] is sorted;
Sorted part of the list: [4, 17, 41, 71]
Unsorted part of the list: [0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:5] is sorted;
Sorted part of the list: [0, 4, 17, 41, 71]
Unsorted part of the list: [7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:6] is sorted;
Sorted part of the list: [0, 4, 7, 17, 41, 71]
Unsorted part of the list: [5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:7] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 41, 71]
Unsorted part of the list: [97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:8] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 41, 71, 97]
Unsorted part of the list: [61, 28, 52]

Breakpoint; invariant: Sublist x[0:9] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 41, 61, 71, 97]
Unsorted part of the list: [28, 52]

Breakpoint; invariant: Sublist x[0:10] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 28, 41, 61, 71, 97]
Unsorted part of the list: [52]
```

Insertion sort

```
def insertion_sort(x):

    for i in range(len(x)):

        j = 0
        element_i = x[i]

        while x[j] < element_i and j < i:
            j += 1

        if i != j:
            x.pop(i)
            x.insert(j, element_i)
```

[Loop invariants]()

Is it true the first time?

If true in one iteration, is it true in the next one?

```
Test list: [41, 17, 71, 4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:1] is sorted;
Sorted part of the list: [41]
Unsorted part of the list: [17, 71, 4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:2] is sorted;
Sorted part of the list: [17, 41]
Unsorted part of the list: [71, 4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:3] is sorted;
Sorted part of the list: [17, 41, 71]
Unsorted part of the list: [4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:4] is sorted;
Sorted part of the list: [4, 17, 41, 71]
Unsorted part of the list: [0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:5] is sorted;
Sorted part of the list: [0, 4, 17, 41, 71]
Unsorted part of the list: [7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:6] is sorted;
Sorted part of the list: [0, 4, 7, 17, 41, 71]
Unsorted part of the list: [5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:7] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 41, 71]
Unsorted part of the list: [97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:8] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 41, 71, 97]
Unsorted part of the list: [61, 28, 52]

Breakpoint; invariant: Sublist x[0:9] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 41, 61, 71, 97]
Unsorted part of the list: [28, 52]

Breakpoint; invariant: Sublist x[0:10] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 28, 41, 61, 71, 97]
Unsorted part of the list: [52]
```

Insertion sort

```
def insertion_sort(x):

    for i in range(len(x)):
        ←—— x[0: i] is sorted
        j = 0
        element_i = x[i]

        while x[j] < element_i and j < i:
            j += 1
        ←—— x[0: j] all smaller than x[i]
        if i != j:
            x.pop(i)
            x.insert(j, element_i)
        ←—— x[0: i+1] is sorted
```

Is it true the first time?

If true in one iteration,
is it true in the next one?

```
Test list: [41, 17, 71, 4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:1] is sorted;
Sorted part of the list: [41]
Unsorted part of the list: [17, 71, 4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:2] is sorted;
Sorted part of the list: [17, 41]
Unsorted part of the list: [71, 4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:3] is sorted;
Sorted part of the list: [17, 41, 71]
Unsorted part of the list: [4, 0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:4] is sorted;
Sorted part of the list: [4, 17, 41, 71]
Unsorted part of the list: [0, 7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:5] is sorted;
Sorted part of the list: [0, 4, 17, 41, 71]
Unsorted part of the list: [7, 5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:6] is sorted;
Sorted part of the list: [0, 4, 7, 17, 41, 71]
Unsorted part of the list: [5, 97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:7] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 41, 71]
Unsorted part of the list: [97, 61, 28, 52]

Breakpoint; invariant: Sublist x[0:8] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 41, 71, 97]
Unsorted part of the list: [61, 28, 52]

Breakpoint; invariant: Sublist x[0:9] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 41, 61, 71, 97]
Unsorted part of the list: [28, 52]

Breakpoint; invariant: Sublist x[0:10] is sorted;
Sorted part of the list: [0, 4, 5, 7, 17, 28, 41, 61, 71, 97]
Unsorted part of the list: [52]
```

Insertion sort

```
def insertion_sort(x):

    for i in range(len(x)):
        ←—— x[0: i] is sorted
        j = 0
        element_i = x[i]

        while x[j] < element_i and j < i:
            j += 1
        ←—— x[0: j] all smaller than x[i]
        if i != j:
            x.pop(i)
            x.insert(j, element_i)
        ←—— x[0: i+1] is sorted
```

Is it true the first time?

If true in one iteration,
is it true in the next one?

Discussion

Would insertion sort qualify as following any of the algorithm design strategies that we have discussed?

Why would that be the case?

Insertion sort

```
def insertion_sort(x):

    for i in range(len(x)):
        ←  x[0: i] is sorted
        j = 0
        element_i = x[i]

        while x[j] < element_i and j < i:
            j += 1
        ←  x[0: j] all smaller than x[i]
        if i != j:
            x.pop(i)
            x.insert(j, element_i)
        ←  x[0: i+1] is sorted
```

Is it true the first time?

**If true in one iteration,
is it true in the next one?**

Discussion

Would insertion sort qualify as following any of the algorithm design strategies that we have discussed?

Why would that be the case?

How about the part where the index j is determined, highlighted in orange?

Insertion sort for a dynamic array: Time efficiency

```
def insertion_sort(x):

    for i in range(len(x)):

        j = 0
        element_i = x[i]

        while x[j] < element_i and j < i:
            j += 1

        if i != j:
            x.pop(i)
            x.insert(j, element_i)
```

Input size *n* given by len(x)

loop executed O(n) times

- O(1) instructions
- Array lookup in O(???) time

- Loop executed O(n) times
  - O(1) instructions

- If x[i] needs to be shifted:
  - O(???) instructions
  - O(???) instructions

Insertion sort for a dynamic array: Time efficiency

```
def insertion_sort(x):

    for i in range(len(x)):

        j = 0
        element_i = x[i]

        while x[j] < element_i and j < i:
            j += 1

        if i != j:
            x.pop(i)
            x.insert(j, element_i)
```

Input size n given by len(x)

loop executed $O(n)$ times

- O(1) instructions
- Array lookup in O(1) time

- **Loop executed $O(n)$ times**
  - **O(1) instructions**

- If x[i] needs to be shifted:
  - $O(n)$ instructions
  - $O(n)$ instructions

$O(n^2)$ time efficiency

Insertion of a new element: Index search

```
def insertion_sort(x):

    for i in range(len(x)):

        j = 0
        element_i = x[i]

        while x[j] < element_i and j < i:
            j += 1

        if i != j:
            x.pop(i)
            x.insert(j, element_i)
```

Could this index be determined more efficiently, given that the sublist x[0: i] is already sorted?

Insertion of a new element: Index search

```
def insertion_sort(x):

  for i in range(len(x)):

    j = 0
    element_i = x[i]

    while x[j] < element_i and j < i:
      j += 1

    if i != j:
      x.pop(i)
      x.insert(j, element_i)
```

Insertion index search problem

Arguments:

- a list of numbers **x**
- an index i such that x[0:i] is sorted
- a number new_element

Find the index where new_element must be inserted so that x remains sorted.

Could this index be determined more efficiently, given that the sublist x[0: i] is already sorted?

Idea: Try divide-and-conquer.

Binary search of the insertion index

We would like to insert new_element = 145 into the following list:

min_index: 0 mid_index: 7 max_index: 14

[**37, 47, 52, 52, 57, 91, 110, <u>117</u>, 118, 147, 151, 158, 167, 195**]

min_index: 8 max_index: 14

Binary search of the insertion index

We would like to insert new_element = 145 into the following list:

min_index: 0 mid_index: 7 max_index: 14

[**37, 47, 52, 52, 57, 91, 110, <u>117</u>, 118, 147, 151, 158, 167, 195**]

min_index: 8 mid_index: 11 max_index: 14

[37, 47, 52, 52, 57, 91, 110, 117, **118, 147, 151, <u>158</u>, 167, 195**]

Binary search of the insertion index

We would like to insert new_element = 145 into the following list:

min_index: 0 mid_index: 7 max_index: 14

[**37, 47, 52, 52, 57, 91, 110, <u>117</u>, 118, 147, 151, 158, 167, 195**]

min_index: 8 mid_index: 11 max_index: 14

[37, 47, 52, 52, 57, 91, 110, 117, **118, 147, 151, <u>158</u>, 167, 195**]

min_index: 8 mid_index: 9 max_index: 11

[37, 47, 52, 52, 57, 91, 110, 117, **118, <u>147</u>, 151**, 158, 167, 195]

min_index: 8 mid_index: 8 max_index: 9

[37, 47, 52, 52, 57, 91, 110, 117, **<u>118</u>**, 147, 151, 158, 167, 195]

Binary search of the insertion index

We would like to insert **new_element = 145** into the following list:

min_index: 0 mid_index: 7 max_index: 14

[**37, 47, 52, 52, 57, 91, 110, <u>117</u>, 118, 147, 151, 158, 167, 195**]

min_index: 8 mid_index: 11 max_index: 14

[37, 47, 52, 52, 57, 91, 110, 117, **118, 147, 151, <u>158</u>, 167, 195**]

min_index: 8 mid_index: 9 max_index: 11

[37, 47, 52, 52, 57, 91, 110, 117, **118, <u>147</u>, 151**, 158, 167, 195]

min_index: 8 mid_index: 8 max_index: 9

[37, 47, 52, 52, 57, 91, 110, 117, **<u>118</u>**, 147, 151, 158, 167, 195]

min_index: 9 max_index: 9

[37, 47, 52, 52, 57, 91, 110, 117, 118, **here!**, 147, 151, 158, 167, 195]

Insertion index binary search

```
def insertion_index_binary_search(x, i, new_element):
    idx_min, idx_max = 0, i
    while idx_max > idx_min:
        idx_mid = (idx_min + idx_max) // 2
        if x[idx_mid] < new_element:

            idx_min = idx_mid+1
        else:

            idx_max = idx_mid

    return idx_min
```

Insertion index binary search

```
def insertion_index_binary_search(x, i, new_element):
    idx_min, idx_max = 0, i
    while idx_max > idx_min:
        idx_mid = (idx_min + idx_max) // 2
        if x[idx_mid] < new_element:

            idx_min = idx_mid+1
        else:

            idx_max = idx_mid

    return idx_min
```

correct index is greater than idx_mid

correct index is smaller than or equal to idx_mid

idx_min = idx_max

Binary search of the insertion index

```
Pre-sorted test list: [0, 28, 71, 81, 107, 155, 263, 351, 459, 521, 587, 658, 663, 700, 705, 761, 775, 799, 833, 8
37, 890, 896, 920, 923, 959, 1075, 1133, 1155, 1207, 1339, 1382, 1461, 1488, 1551, 1552, 1594, 1743, 1854, 1877, 1
907, 1907, 2000, 2038, 2120, 2127, 2152, 2233, 2234, 2459, 2478]
New element: 1140

Evaluating x[0:50] = [0, 28, 71, 81, 107, 155, 263, 351, 459, 521, 587, 658, 663, 700, 705, 761, 775, 799, 833, 83
7, 890, 896, 920, 923, 959, 1075, 1133, 1155, 1207, 1339, 1382, 1461, 1488, 1551, 1552, 1594, 1743, 1854, 1877, 19
07, 1907, 2000, 2038, 2120, 2127, 2152, 2233, 2234, 2459, 2478]
Middle index 25 with x[25] = 1075

Evaluating x[26:50] = [1133, 1155, 1207, 1339, 1382, 1461, 1488, 1551, 1552, 1594, 1743, 1854, 1877, 1907, 1907, 2
000, 2038, 2120, 2127, 2152, 2233, 2234, 2459, 2478]
Middle index 38 with x[38] = 1877

Evaluating x[26:38] = [1133, 1155, 1207, 1339, 1382, 1461, 1488, 1551, 1552, 1594, 1743, 1854]
Middle index 32 with x[32] = 1488

Evaluating x[26:32] = [1133, 1155, 1207, 1339, 1382, 1461]
Middle index 29 with x[29] = 1339

Evaluating x[26:29] = [1133, 1155, 1207]
Middle index 27 with x[27] = 1155

Evaluating x[26:27] = [1133]
Middle index 26 with x[26] = 1133

Index 27 specified for insertion
```

Discussion

What is the time efficiency of this binary search?

Binary search of the insertion index

```
Pre-sorted test list: [0, 28, 71, 81, 107, 155, 263, 351, 459, 521, 587, 658, 663, 700, 705, 761, 775, 799, 833, 8
37, 890, 896, 920, 923, 959, 1075, 1133, 1155, 1207, 1339, 1382, 1461, 1488, 1551, 1552, 1594, 1743, 1854, 1877, 1
907, 1907, 2000, 2038, 2120, 2127, 2152, 2233, 2234, 2459, 2478]
New element: 1140

Evaluating x[0:50] = [0, 28, 71, 81, 107, 155, 263, 351, 459, 521, 587, 658, 663, 700, 705, 761, 775, 799, 833, 83
7, 890, 896, 920, 923, 959, 1075, 1133, 1155, 1207, 1339, 1382, 1461, 1488, 1551, 1552, 1594, 1743, 1854, 1877, 19
07, 1907, 2000, 2038, 2120, 2127, 2152, 2233, 2234, 2459, 2478]
Middle index 25 with x[25] = 1075

Evaluating x[26:50] = [1133, 1155, 1207, 1339, 1382, 1461, 1488, 1551, 1552, 1594, 1743, 1854, 1877, 1907, 1907, 2
000, 2038, 2120, 2127, 2152, 2233, 2234, 2459, 2478]
Middle index 38 with x[38] = 1877

Evaluating x[26:38] = [1133, 1155, 1207, 1339, 1382, 1461, 1488, 1551, 1552, 1594, 1743, 1854]
Middle index 32 with x[32] = 1488

Evaluating x[26:32] = [1133, 1155, 1207, 1339, 1382, 1461]
Middle index 29 with x[29] = 1339

Evaluating x[26:29] = [1133, 1155, 1207]
Middle index 27 with x[27] = 1155

Evaluating x[26:27] = [1133]
Middle index 26 with x[26] = 1133

Index 27 specified for insertion
```

Discussion

What is the time efficiency of this binary search?

Could this algorithm also be used to find **whether a sorted list contains a certain value**, and to return the index for that value if it does?

Improved insertion sort algorithm

```
def insertion_sort(x):

    for i in range(len(x)):

        j = 0
        element_i = x[i]

        j = insertion_index_binary_search( \
                x, i, element_i, False )

        if i != j:
            x.pop(i)
            x.insert(j, element_i)
```

Input size *n* given by len(x)

loop executed O(*n*) times

- O(1) instructions
- Array lookup in O(1) time
- **Improved due to binary search**
- **If x[i] needs to be shifted:**
  - **O(*n*) instructions**
  - **O(*n*) instructions**

O(n^2) average/worst case time efficiency

Improved insertion sort algorithm

```
def insertion_sort(x):
    for i in range(len(x)):
        j = 0
        element_i = x[i]

        j = insertion_index_binary_search( \
                x, i, element_i, False )

        if i != j:
            x.pop(i)
            x.insert(j, element_i)
```

Input size n given by len(x)

loop executed O(n) times

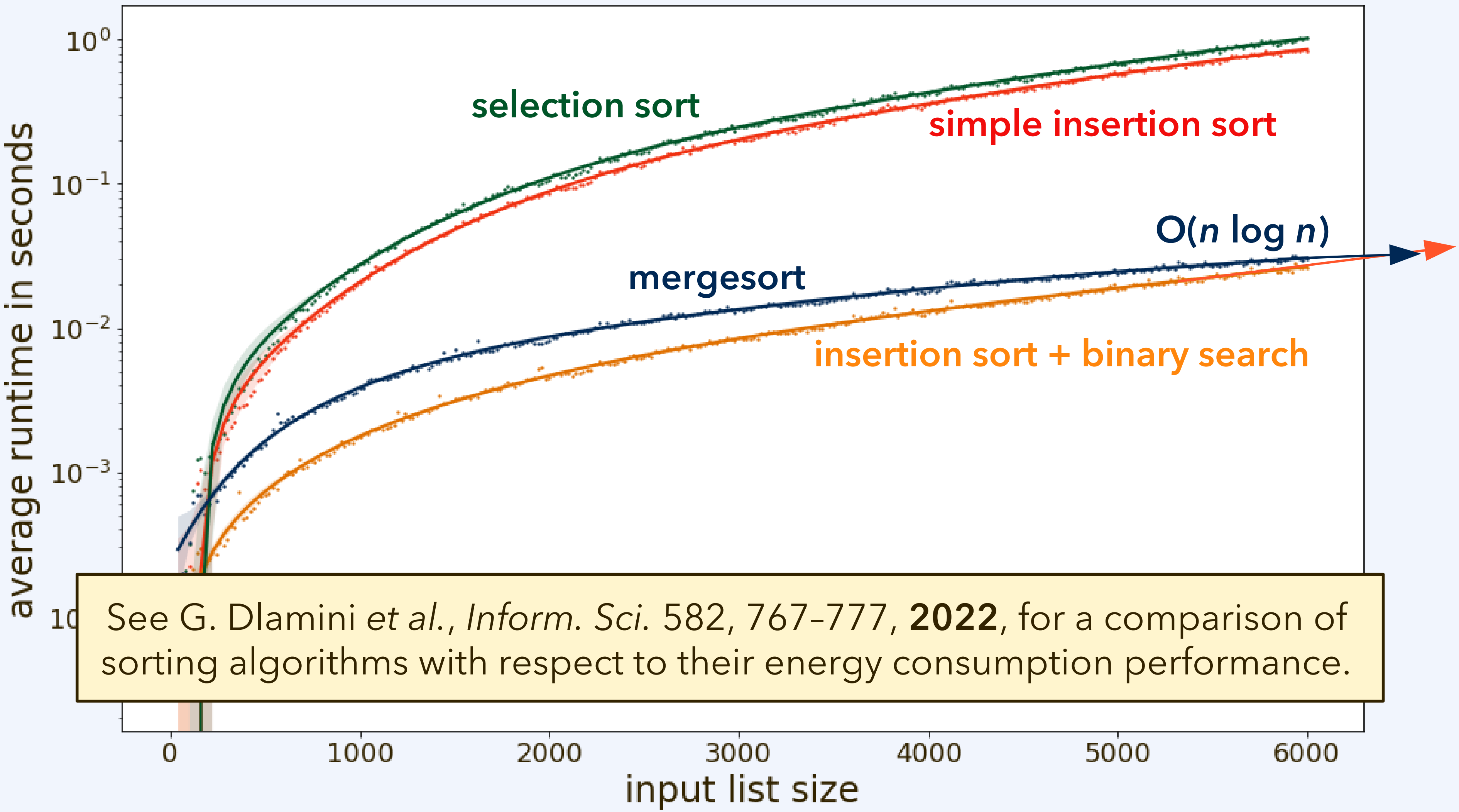
- O(1) instructions
- Array lookup in O(1) time
- **Improved due to binary search**
- **If x[i] needs to be shifted:**
  - **O(n) instructions**
  - **O(n) instructions**

What is the best-case time efficiency?

O(n^2) average/worst case time efficiency

Sorting algorithms: Overview

Sorting algorithms: Overview



See G. Dlamini *et al.*, *Inform. Sci.* 582, 767–777, **2022**, for a comparison of sorting algorithms with respect to their energy consumption performance.

Tree data structures: Introduction

CO2412

23rd November 2021

Binary search: Unapplicable to linked lists

We would like to insert **new_element = 145** into the following list:

min_index: 0 mid_index: 7 max_index: 14

[**37, 47, 52, 52, 57, 91, 110, 117, 118, 147, 151, 158, 167, 195**]

min_index: 8 mid_index: 11 max_index: 14

[37, 47, 52, 52, 57, 91, 110, 117, **118, 147, 151, 158, 167, 195**]

min_index: 8 mid_index: 9 max_index: 11

[37, 47, 52, 52, 57, 91, 110, 117, **118, 147, 151**, 158, 167, 195]

min_index: 8 mid_index: 8 max_index: 9

[37, 47, 52, 52, 57, 91, 110, 117, **118**, 147, 151, 158, 167, 195]

min_index: 9 max_index: 9

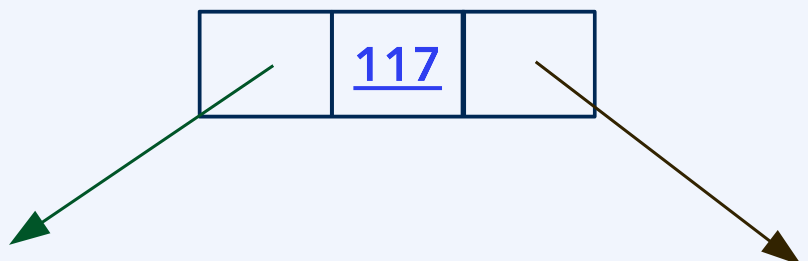
[37, 47, 52, 52, 57, 91, 110, 117, 118, **here!**, 147, 151, 158, 167, 195]

Question: Why does this work for a dynamic array, but not for a linked list?

Non-sequential linked data structures

What if we design a linked data structure to recover the binary search feature?

[37, 47, 52, 53, 57, 91, 110, **117**, 118, 147, 151, 158, 167, 195]



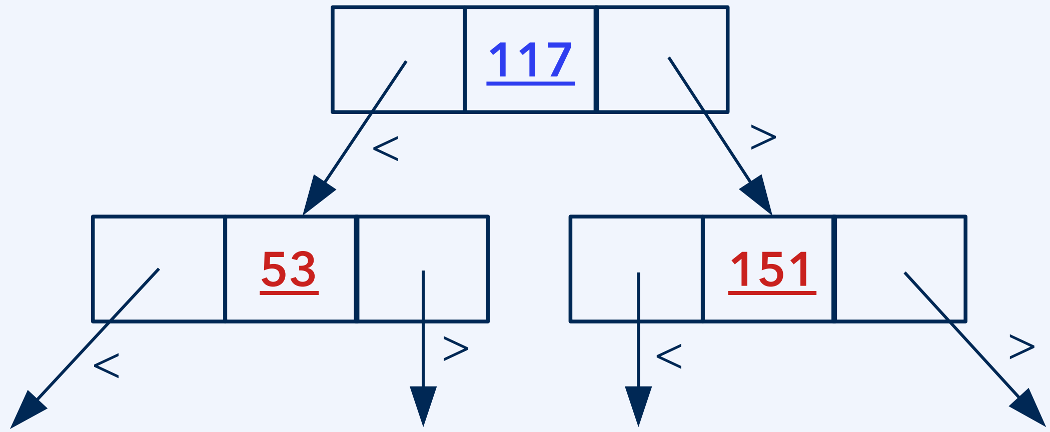
Is your value smaller than the root element? Then go here …

Is your value greater than the root element? Then go here …

Non-sequential linked data structures

What if we design a linked data structure to recover the binary search feature?

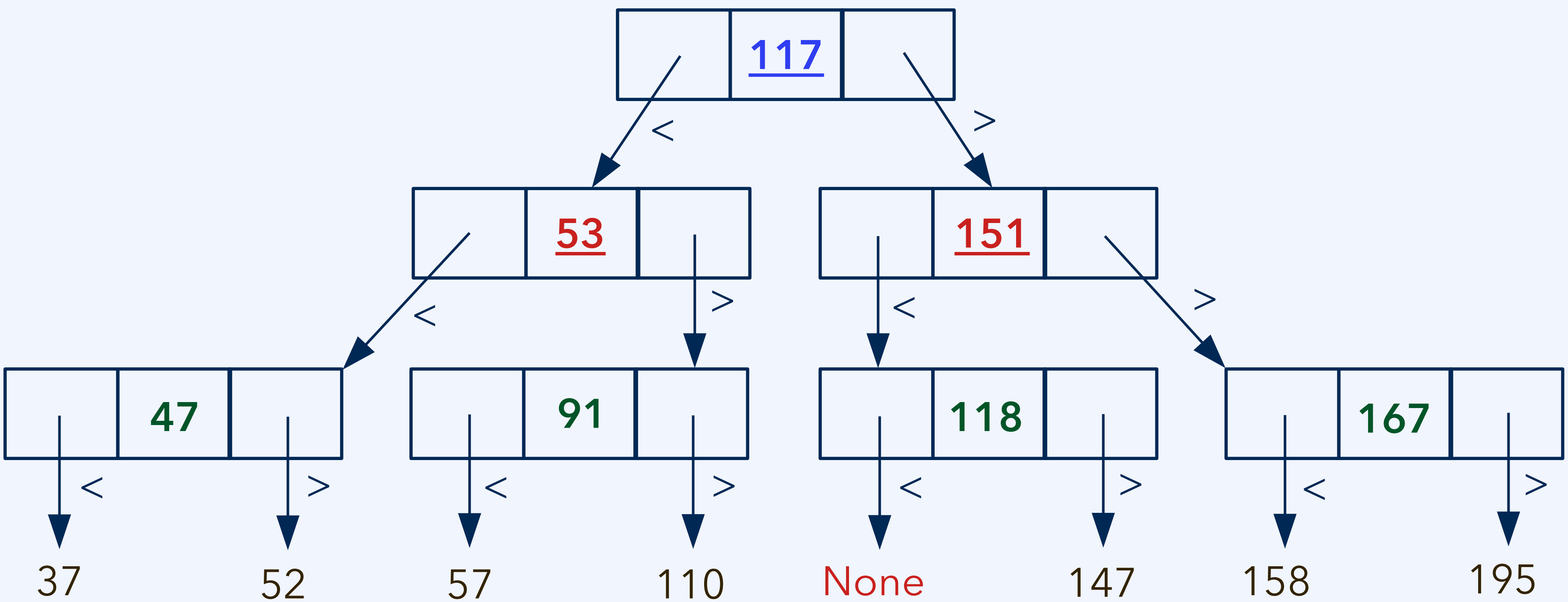
[37, 47, 52, **53**, 57, 91, 110, **117**, 118, 147, **151**, 158, 167, 195]



Tree data structures

What if we design a linked data structure to recover the binary search feature?

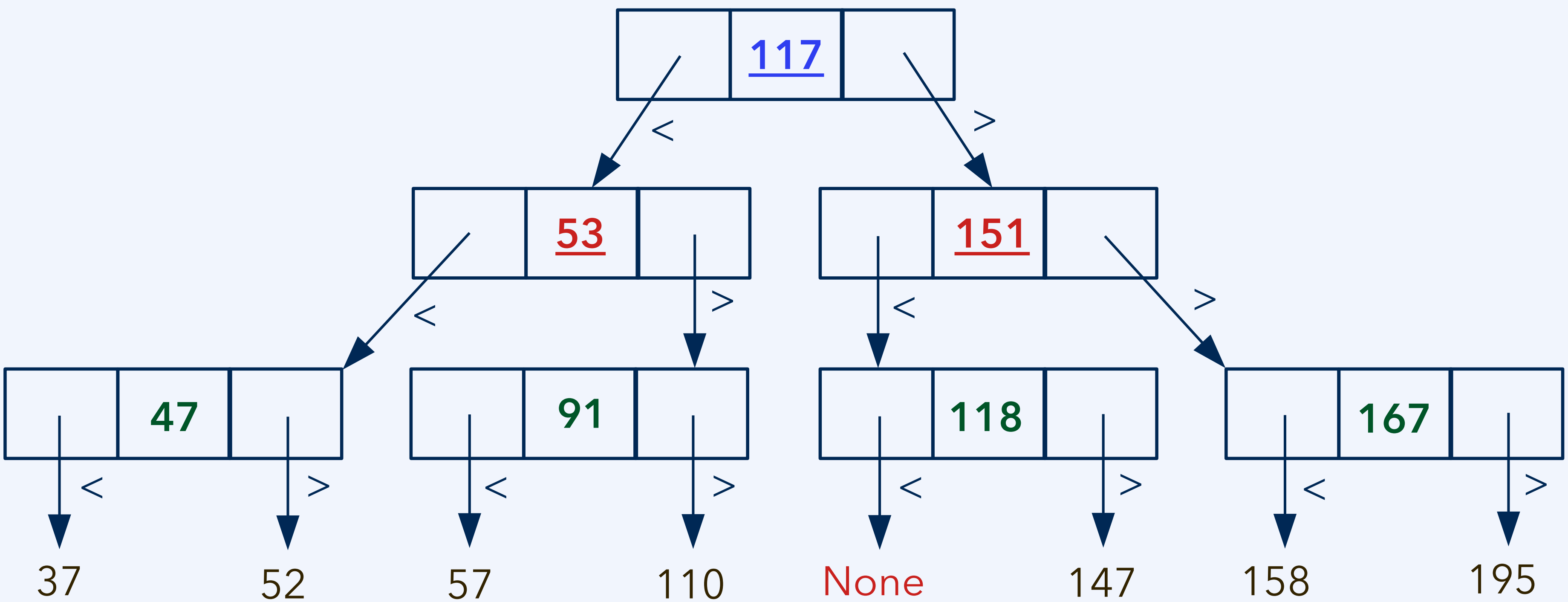
[37, **47**, 52, <u>**53**</u>, 57, **91**, 110, <u>**117**</u>, **118**, 147, <u>**151**</u>, 158, **167**, 195]



Tree data structures: Binary search trees

This data structure is known as a **binary search tree.**

[37, **47**, 52, **53**, 57, **91**, 110, **117**, **118**, 147, **151**, 158, **167**, 195]



University of
Central Lancashire
UCLan

CO2412
Computational Thinking

Arrays and linked lists: Overview
Sorting: Overview
Tree data structures: Introduction

Where opportunity creates success