Fundamental of Computer Science
CS1FC16: Lecture 05

Data Structure

Dr Varun Ojha

Department of Computer Science

University of Reading

Learning Objectives

On completion of three parts of this lecture, you will be able to

- Understand linear and non-linear data structures: array (list), linked list, stacks, queues, trees.
- Evaluate a tree data structure to perform *search*
- Solve expression tree

Content of this lecture

- Linear data structure
  - Linked list
  - Stacks
  - Queue
  - Circular buffer
- Non-linear data structure
  - Tree
  - Arithmetic operations
- Exercises

Fundamental of Computer Science
CS1FC16: Lecture 05, Part – I

Linear Data structure

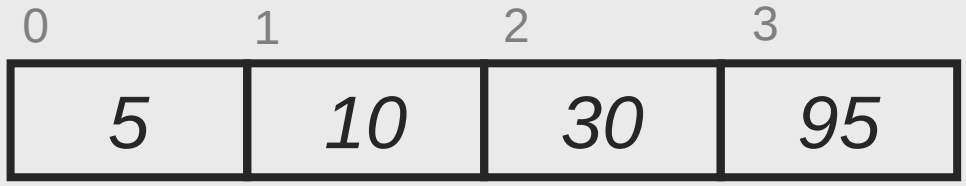
Dr Varun Ojha

Department of Computer Science

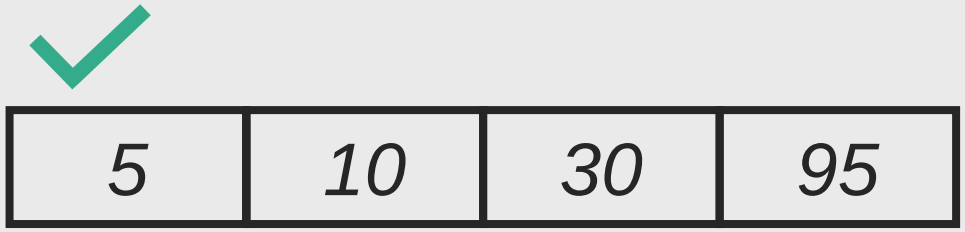
University of Reading

Array

- An *Array* is a collection of elements (data), each of which is indexed contiguously, e.g.:

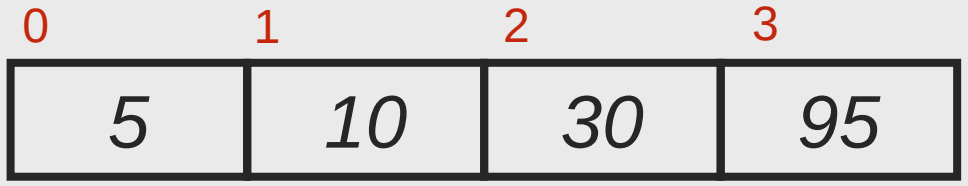


- An array is a *homogenous* data structure, meaning all elements in an array are of the same type, e.g.:



Array

- *Accessing* an element from array takes constant time $O(1)$
- Lets A is our array as follows



- A[2] produce an element (data) 30 only in 1 unit operation that takes a constant time, c sec.
- Similarly, *search / replace* takes a constant time $O(1)$
- Array does not allow *deletion* operation

Node

- A *node* in a data structure is an object that holds data and pointer(s), i.e., reference(s) to other object(s).

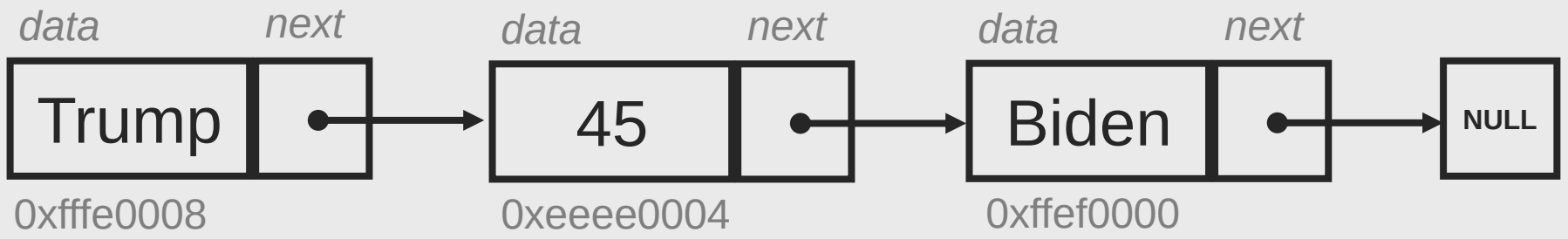
```
typedef struct list_node list;
struct list_node {
    elem data;
    list* pointer;
};
```

Node

- Node of a linked list data structure has:
  - The *data* (stored element)
  - The *next* is a pointer (reference) to memory of next node

Single Linked List

- A *single-linked list* is a sequence of nodes, a collection of objects, each of which points to its successor, with the last node having the NULL pointer
- The null pointer does not point anywhere, following it is an error
- The *empty list*, with no elements, is sometimes called the *null list*

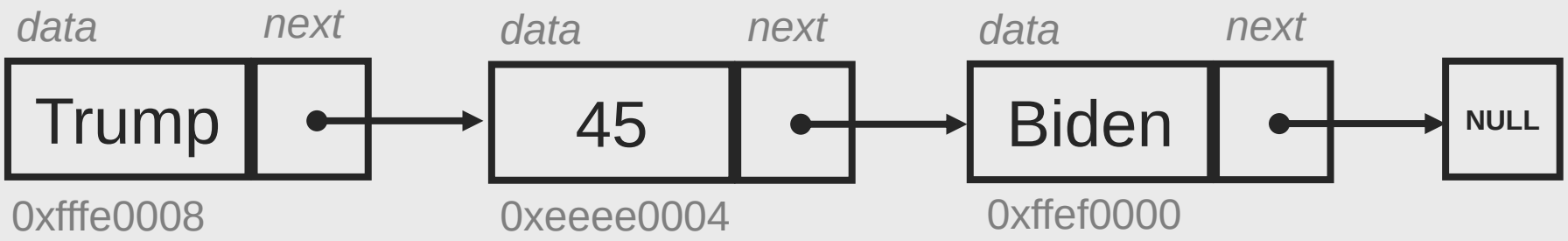


Head and Tail

- The element of the first node of a list is called the head
- The remaining nodes of a list are called the tail

```
[Trump 45 Biden] -> list;

Head(list) -> Trump

Tail(list) -> [45 Biden]
```

- It is an error to take the head or tail of an empty list



Why Linked List

Unlike Array, Linked list allows:

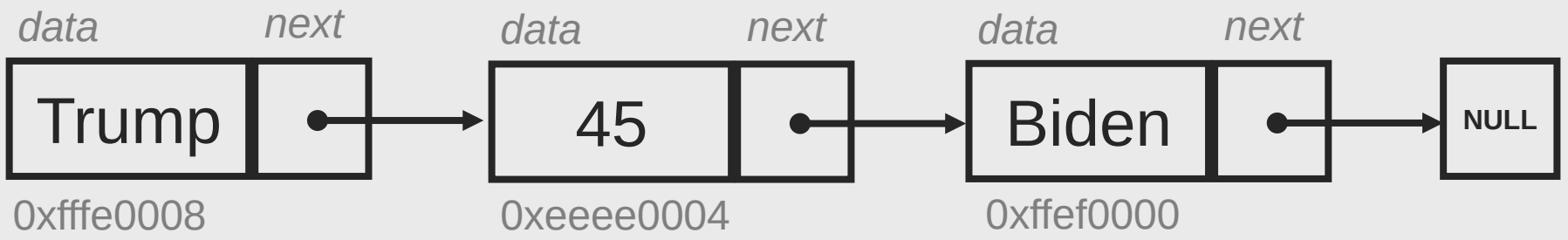
- Elements to be stored at non-contiguous memory locations, leading to *storage efficiency*. However, access to data takes $O(n)$ time.
- Insertion and deletion operation
- Storage of *homogenous* data, as well as, *heterogenous* data

Constructing a Linked List

- Constructing a list by adding elements to the head of the list takes O(1)

```
typedef struct list_node list;
struct list_node {
    elem data;
    list* next;
};
```

```
list* head = alloc(list);
head->data = “Trump”;
head->next = alloc(list)
head->next->data = 45
head->next->next = alloc(list)
head->next->next->data = “Biden”
head->next->next->next = NULL
```

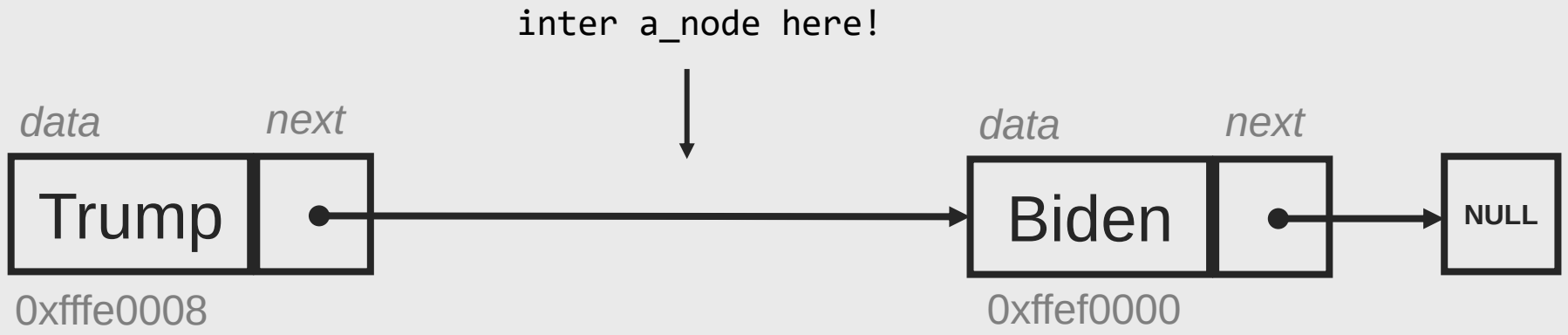


Insertion to a Linked List

- Constructing a list by adding elements to the head of the list takes O(1)

```
list* head = alloc(list);
head->data = “Trump”;
list* b_node = alloc(list)
b_node->data = “Biden”

head->next = b_node
b_node->next = NULL
```

```
list* a_node = alloc(list)
a_node->data = 45
```



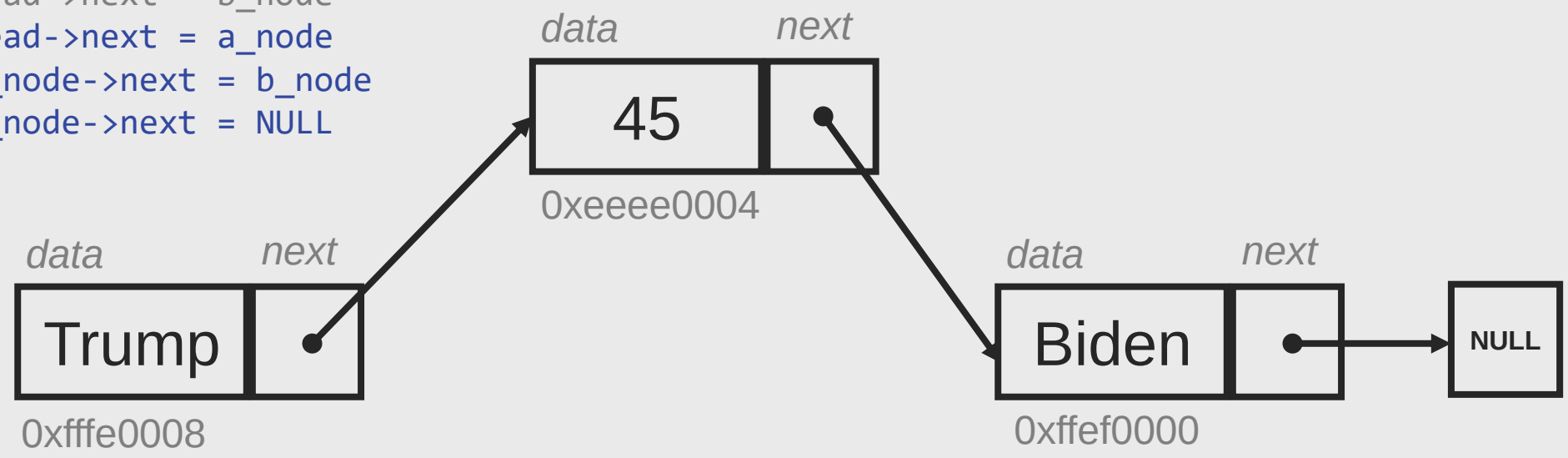
Insertion to a Linked List

- Constructing a list by adding elements to the head of the list takes O(1)

```
list* head = alloc(list);
head->data = “Trump”;
list* b_node = alloc(list)
b_node->data = “Biden”
```

```
list* a_node = alloc(list)
a_node->data = 45
```

```
~~head->next = b_node~~
head->next = a_node
a_node->next = b_node
b_node->next = NULL
```



Deletion to a Linked List

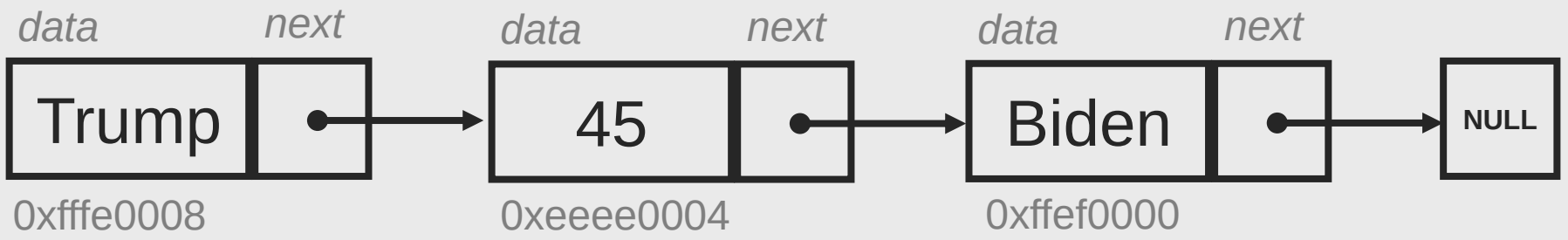
- Constructing a list by adding elements to the head of the list takes O(1)

```
list* head = alloc(list);
head->data = “Trump”;
```

```
list* a_node = alloc(list)
a_node->data = 45
```

```
list* b_node = alloc(list)
b_node->data = “Biden”
```

```
head->next = a_node
a_node->next = b_node
b_node->next = NULL
```



Deletion to a Linked List

- Constructing a list by adding elements to the head of the list takes O(1)

```
list* head = alloc(list);
head->data = “Trump”;
```

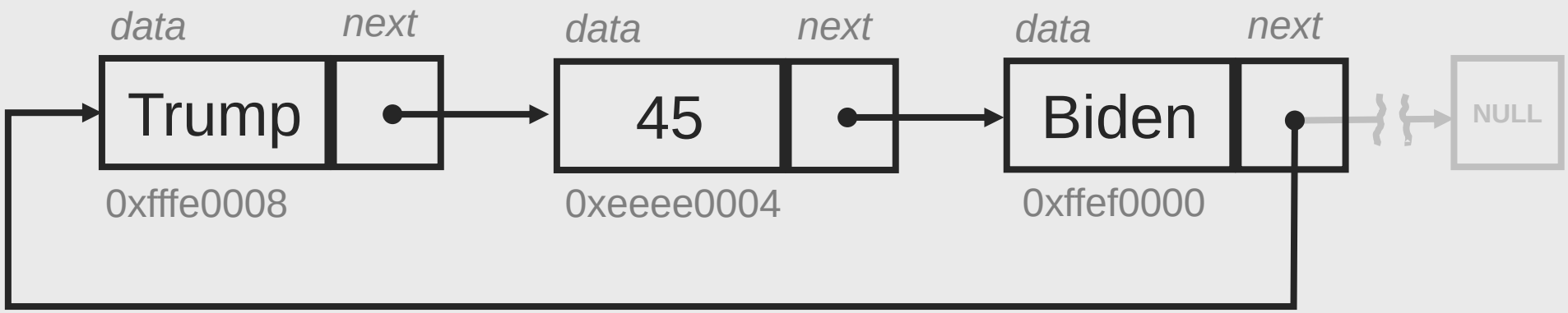
```
list* a_node = alloc(list)
a_node->data = 45
```

```
list* b_node = alloc(list)
b_node->data = “Biden”
```

```
head->next = a_node
head->next = b_node
b_node->next = NULL
```

Special List

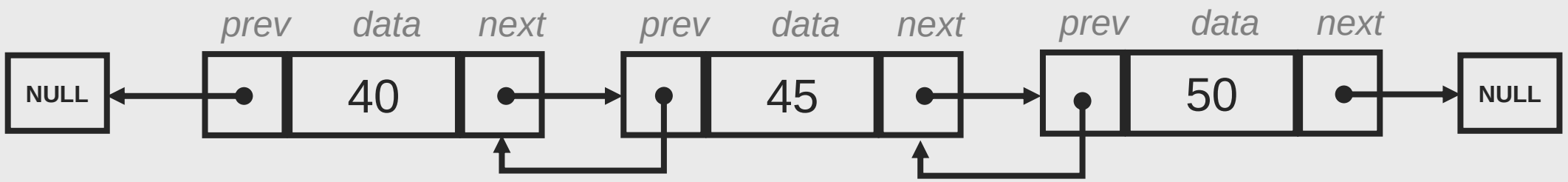
- Circular Linked List
  - last node refers to its first



Special List

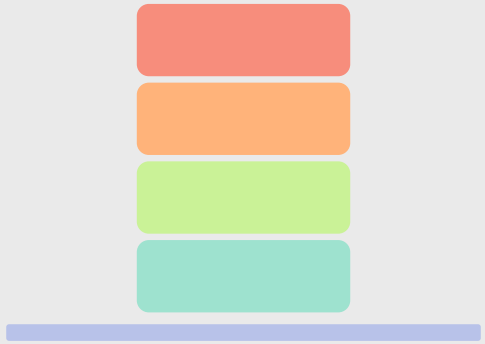
- Double-Linked Lists
  - node has data, previous and next node reference

```
typedef struct list_node list;
struct list_node {
    elem data;
    list* next;
    list* previous;
};
```



Double-Linked Lists

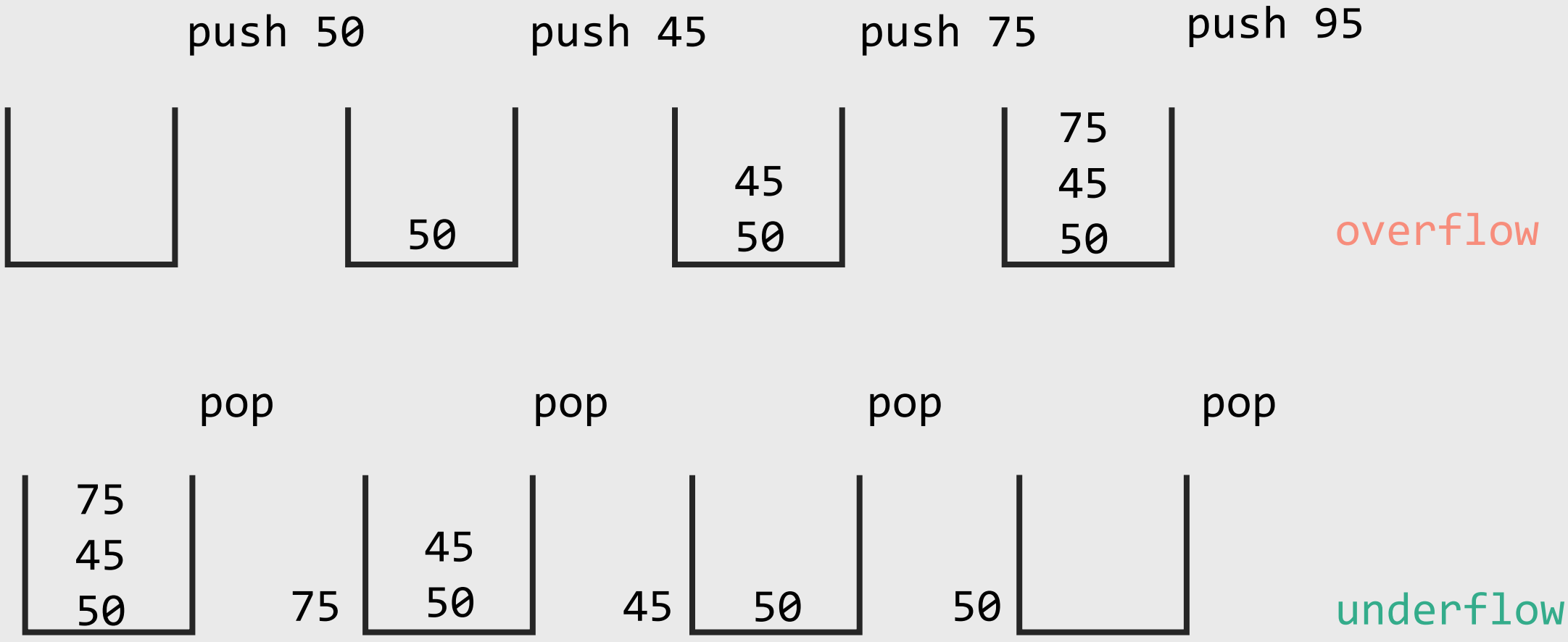
- An element can be pushed or popped onto either end of a double-linked list in time $O(1)$
- A double-linked list can be pushed or popped onto either end of another double-linked list in time $O(1)$
- An element can be inserted into or deleted from an indexed location in a double-linked list of length in worst-case time $O(n/2)$
- The corresponding worst-case time for a single-linked list is time $O(n)$
- But the extra link costs memory



Stacks

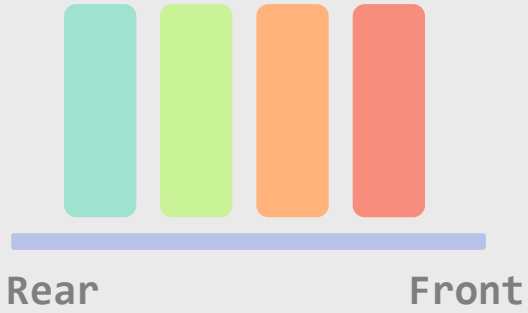
- An empty stack has no elements
- The empty element is denoted by epsilon:
- Data is *pushed* onto a stack and is *popped* off a stack
- A stack is cleared by discarding all elements until it is empty
- It is an *underflow* error to pop an element off an empty stack
- It is an *overflow* error if there is not enough memory to push an element onto a stack

Stacks


push 50
push 45
push 75
push 95
50
45
50
75
45
50
overflow
pop
pop
pop
pop
75
45
50
75
45
50
45
50
50
underflow

Stacks

- Stacks can be implemented in software using lists
- The empty list is the empty stack
- Constructing a list by adding an element as head pushes the element onto the stack
- Destroying the list by returning its head and tail pops the head off the stack, leaving the tail as the shortened stack
- Stacks implement last-in-first-out, LIFO, behaviour



Queues

- Queues used to preserve the order of elements in a generate and test algorithm
- Queues may be used to process real-time events in sequence
- Queues may be used to enforce first-in-first-out, FIFO behaviour
- Pipes are FIFOs
- Almost all modern computers are pipelined
- A queue can be implemented as a double-linked list
- Time complexity to search an element takes $O(n)$

Fundamental of Computer Science
CS1FC16: Lecture 05, Part – II

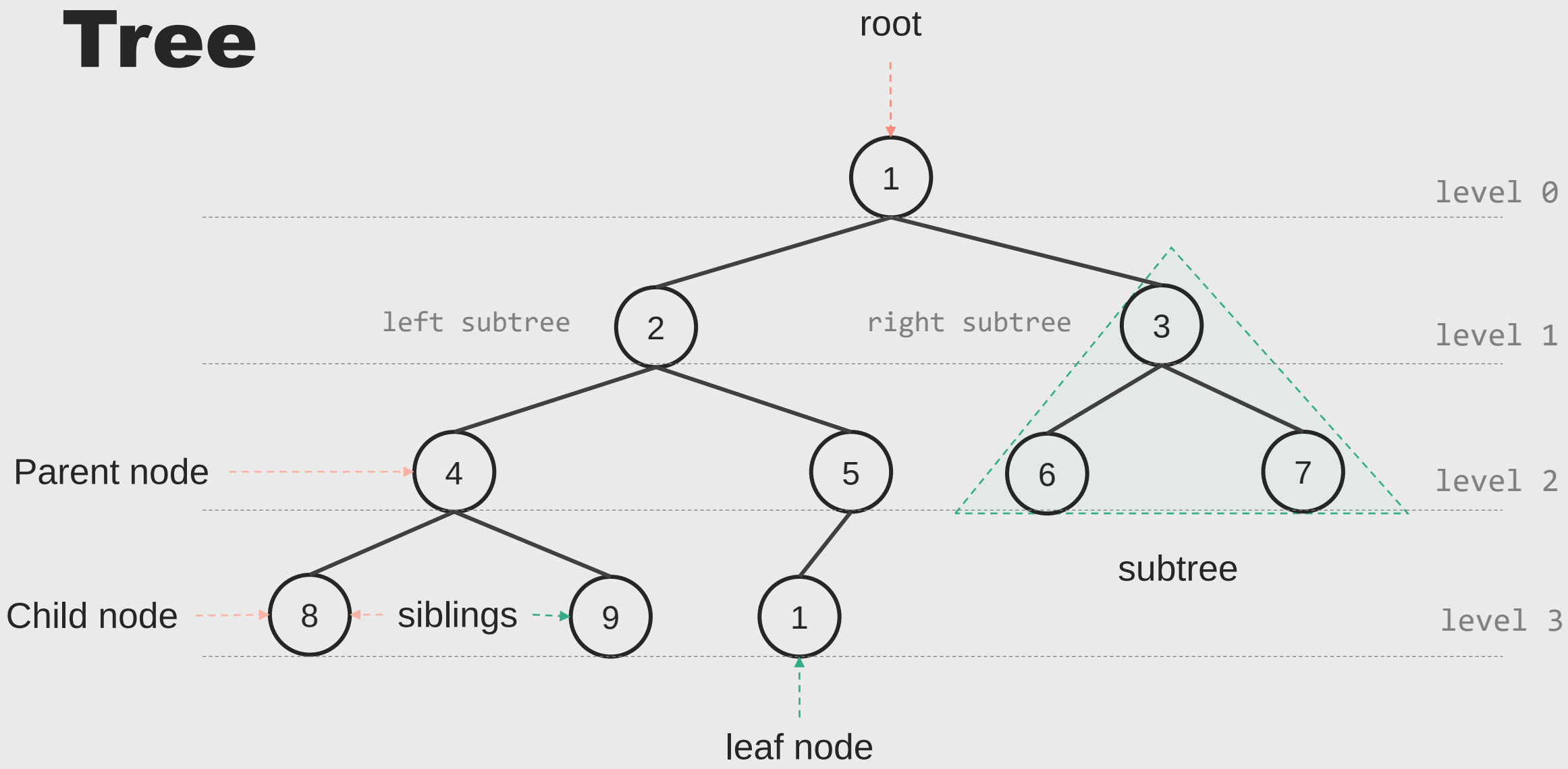
Non-Linear Data Structure

Dr Varun Ojha

Department of Computer Science

University of Reading

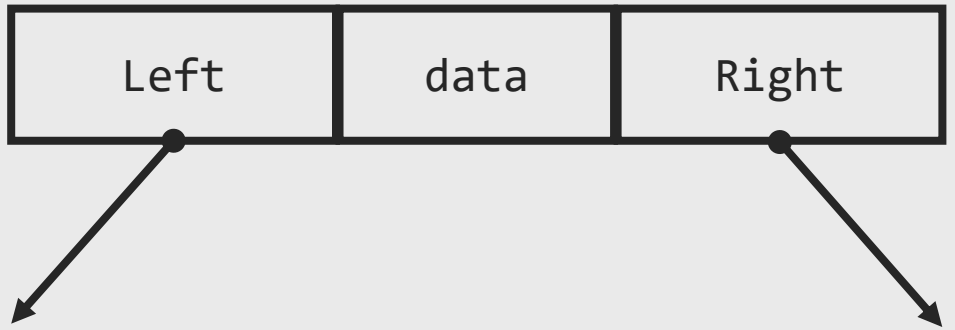
Tree


root
1
level 0
left subtree
2
right subtree
3
level 1
Parent node
4
5
6
7
level 2
subtree
Child node
8
siblings
9
1
level 3
leaf node

Node

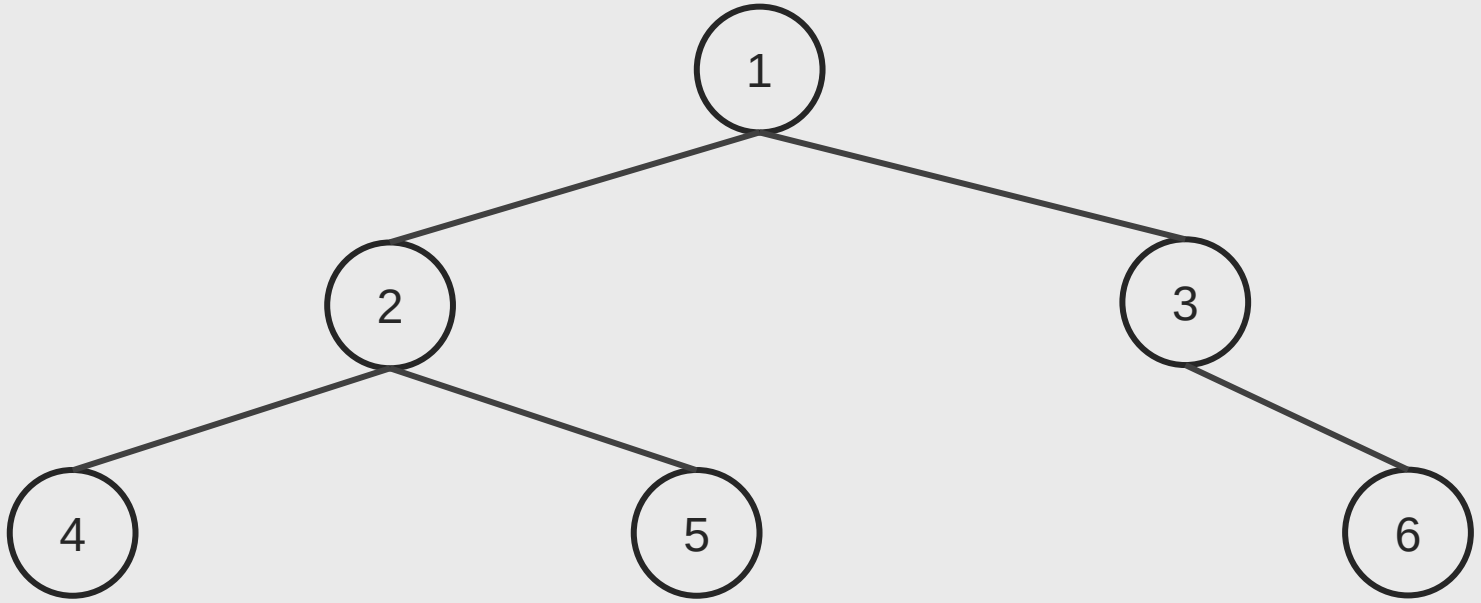
- Node of a tree data structure has:
  - The *data* (stored element)
  - The *LeftChild* is a pointer (reference) to memory of left subtree
  - The *RightChild* is a pointer (reference) to memory of right subtree

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```



Trees

```
/*  [ value [ left ]  [ right ]  ]  */
T -> [1 [2 [4 [] []]  [5 [] []]   [3 [] [6 [] []]]]
```



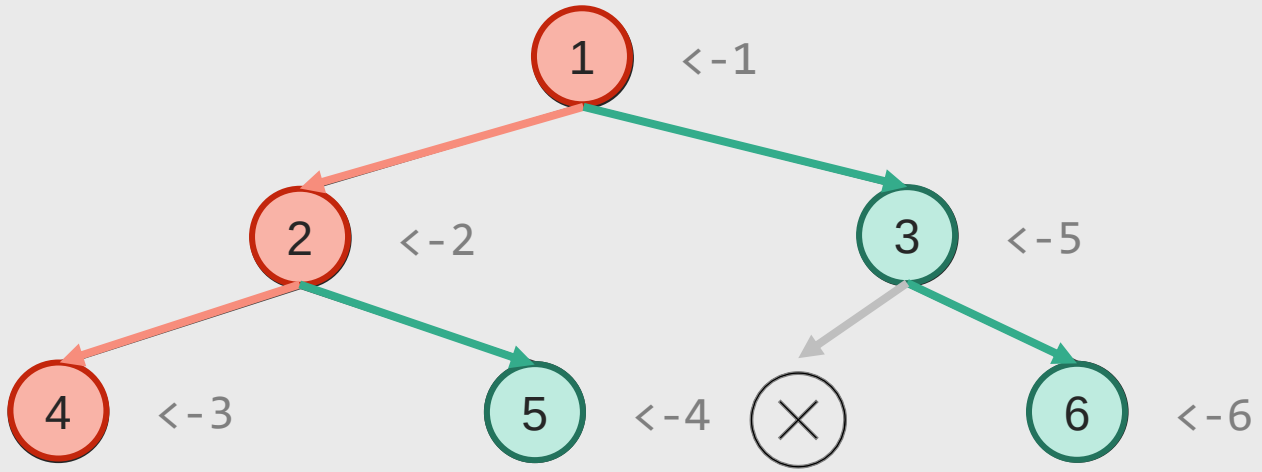
Pre-Order Search

```
/* pr is a printing subroutine. */

vars Value = 1, Left = 2, Right = 3;

define preorder(list);
    unless list = [] then
        pr (list(Value))      /* Value */
        preorder(list(Left )) /* Left */
        preorder(list(Right)) /* Right */
    endunless
enddefine


preorder(T);
```

```
124536
```

Pre-Order Search

- Processes **Value** then Left then Right



1 2 4 5 3 6

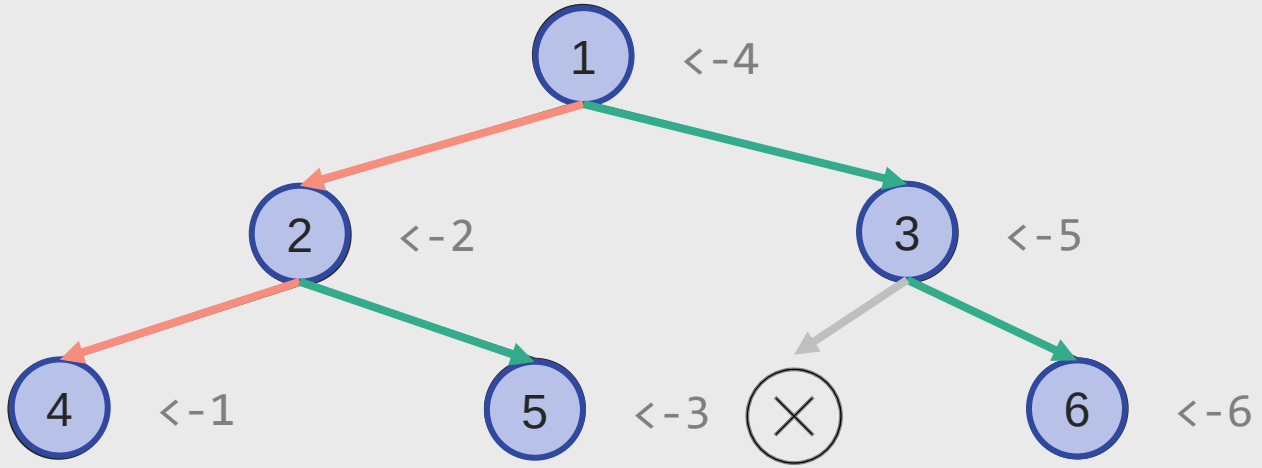
In-Order Search

```
/* pr is a printing subroutine. */

vars Value = 1, Left = 2, Right = 3;

define inorder(list);
    unless list = [] then
        inorder(list(Left )) /* Left */
        pr (list(Value))     /* Value */
        inorder(list(Right)) /* Right */
    endunless
enddefine


inorder(T);
```

```
124536
```

In-Order Search

- Processes Left then **Value** then Right



4 2 5 1 3 6

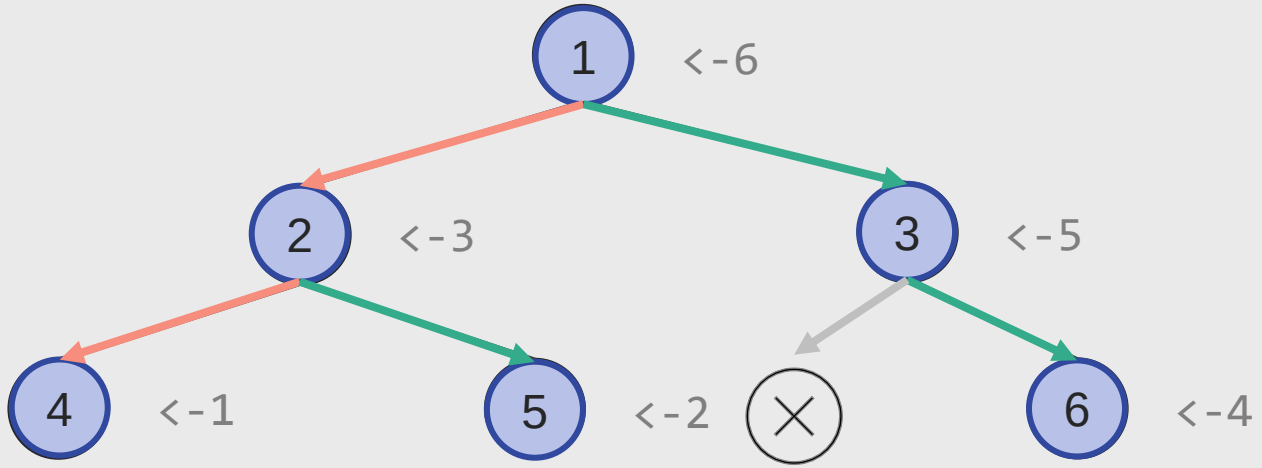
Post-Order Search

```
/* pr is a printing subroutine. */

vars Value = 1, Left = 2, Right = 3;

define postorder(list);
    unless list = [] then
        postorder(list(Left )) /* Left */
        postorder(list(Right)) /* Right */
        pr (list(Value))       /* Value */
    endunless
enddefine


postorder(T);
```

```
124536
```

Post-Order Search

- Processes Left then Right then **Value**



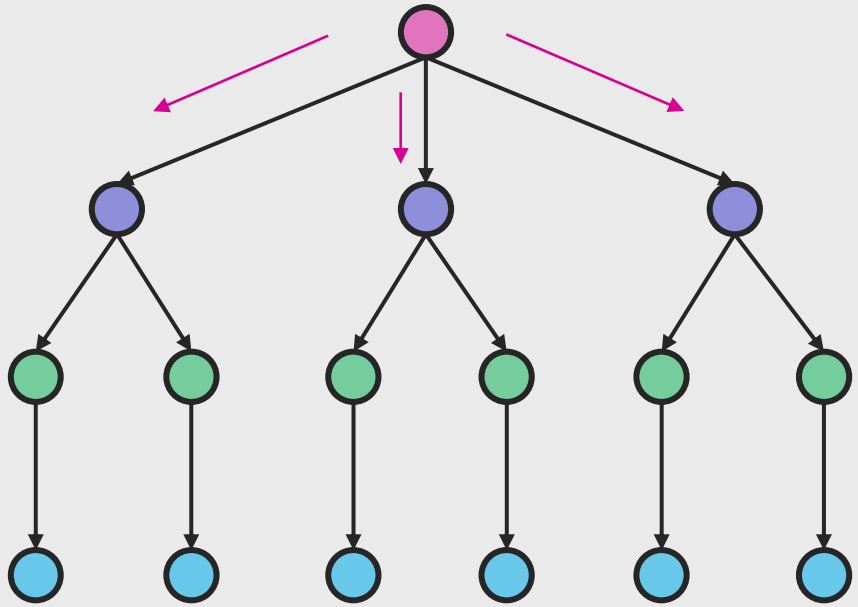
4 5 2 6 3 1

Breadth-First Search

- Breadth-First search (BFS) expands nodes in order of their depth from the root.
- Implemented by first-in first-out (FIFO) queue.
- BFS will find a shortest path to a goal.
- Time/Space Complexity - **branching factor b** and the solution depth d.
- Generate all the nodes up to level d.
- Total number of nodes in BFS

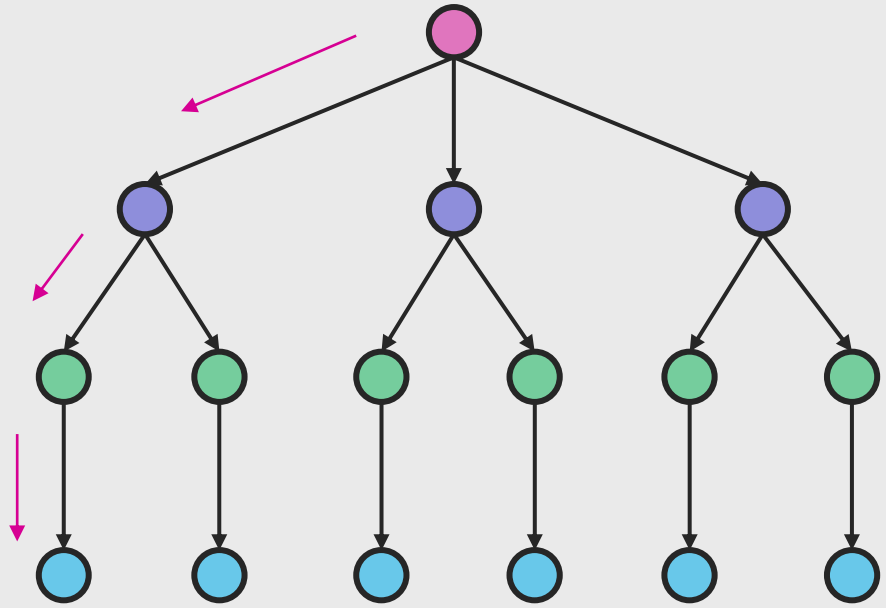
$$1 + b + b^2 + \cdots + bd = O(b^d)$$

- BFS **will exhaust the memory in minutes**.



Depth-First Search

- **Depth-First is iterative-deepening**
  - First performs a DFS to depth one. Than starts over executing DFS to depth two and so on.
- Implemented by LIFO stack
- Space Complexity is linear in the maximum search depth.
- DFS generate the same set of nodes as BFS
- Time Complexity is $O(b^d)$
- **The first solution DFS found may not be the optimal one**.
- **On infinite (branch) tree DFS may not terminate**.

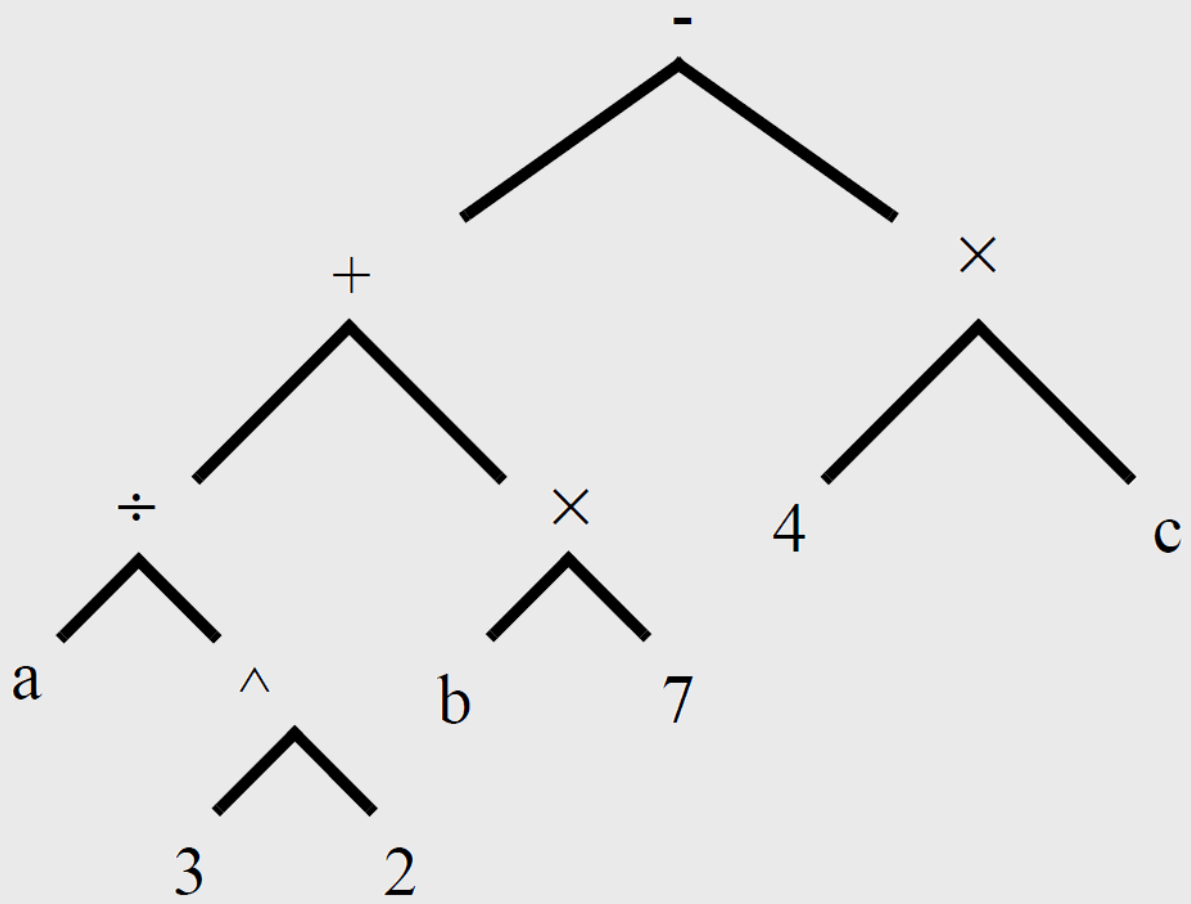


Expression Tree

- Expresion

  x = a ÷ 3^2 + b × 7 - 4 × c

  - Pre-fix Arithmetic
  - In-fix Arithmetic
  - Post-fix Arithmetic



Pre-fix Arithmetic

- **Expression**

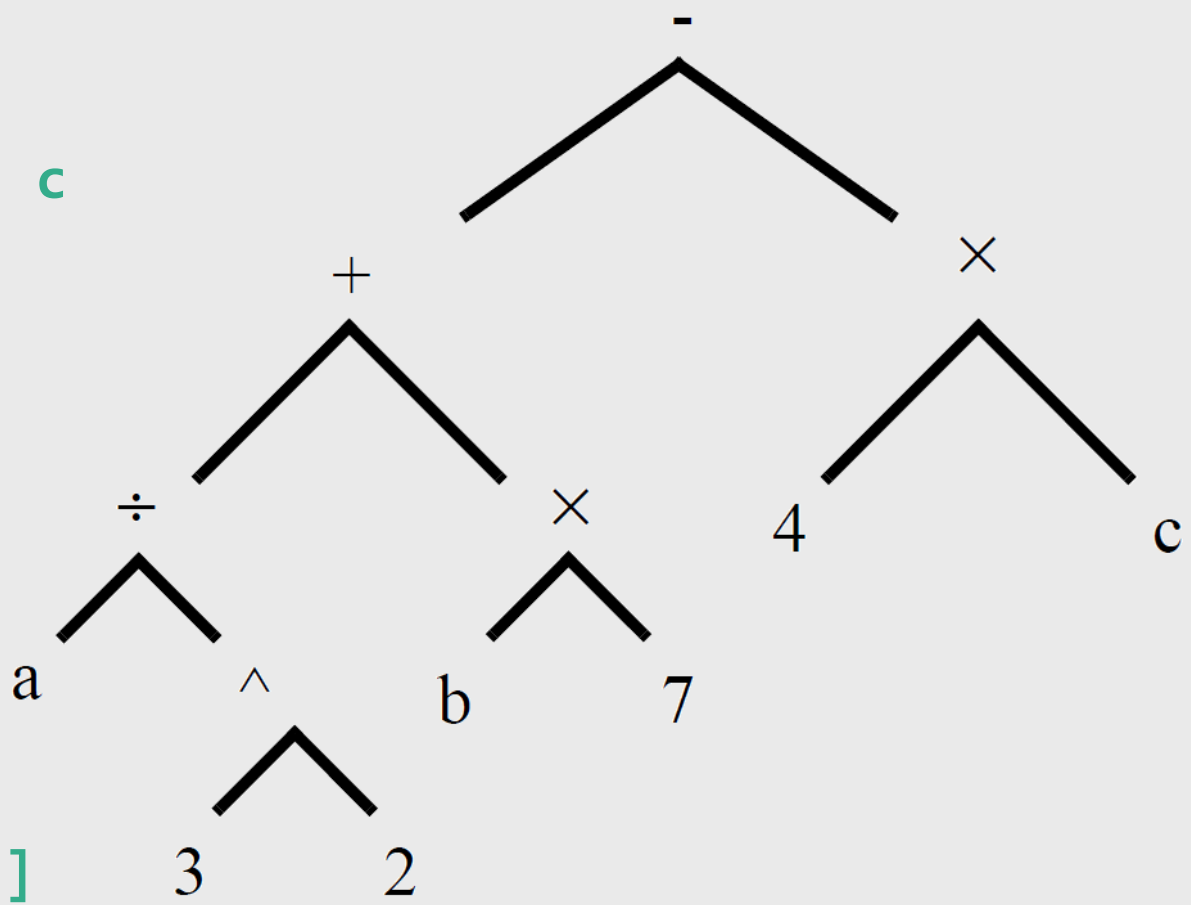
  `x = a ÷ 3^2 + b × 7 - 4 × c`

- **Pre-fix Arithmetic**

  `=`

- **Pre-Order Tree Search**

  `[ - + ÷ a ^3 2 × b 7 × 4 c ]`



In-fix Arithmetic

- **Expression**

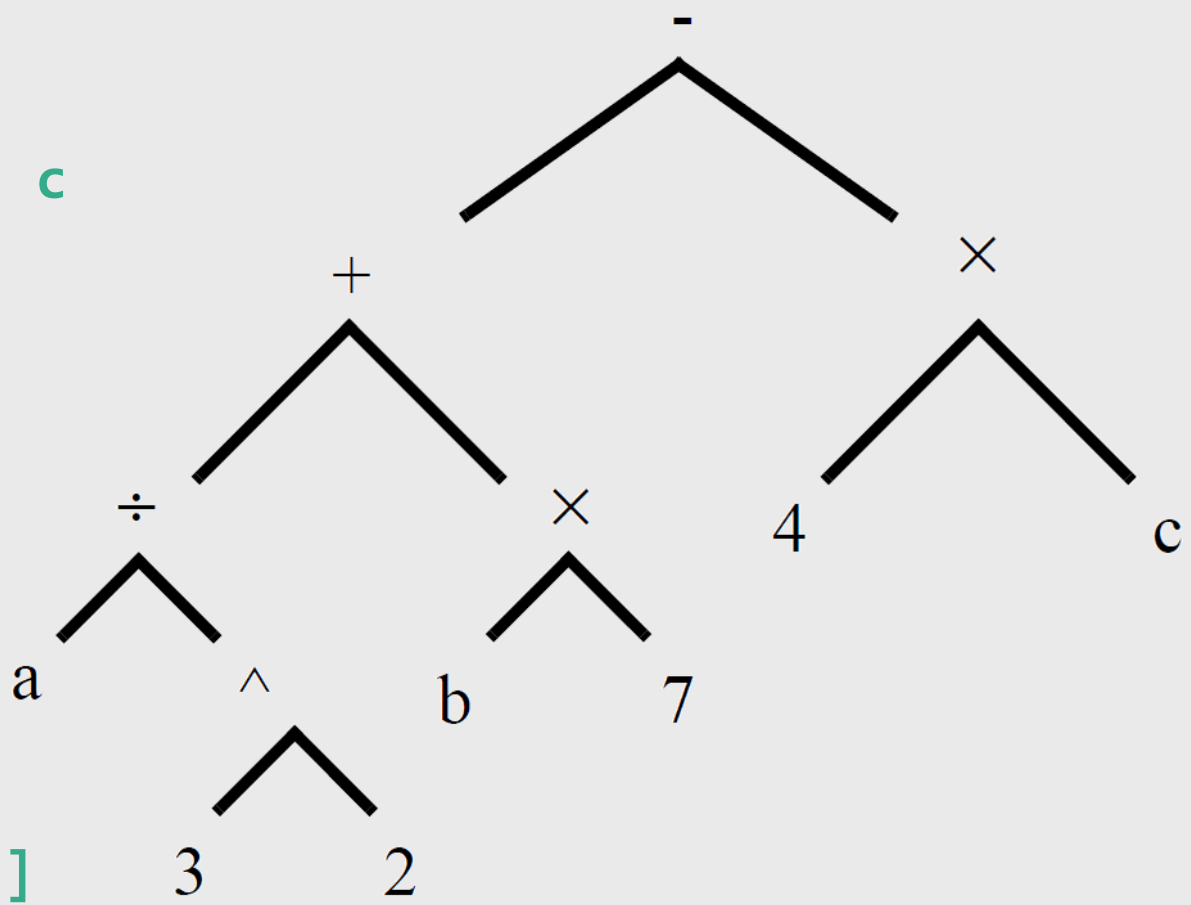
  x = a ÷ 3^2 + b × 7 - 4 × c

- In-fix Arithmetic

  =

- In-Order Tree Search

  [a ÷ 3^2 + b × 7 - 4 × c]



Post-fix Arithmetic

- Expression

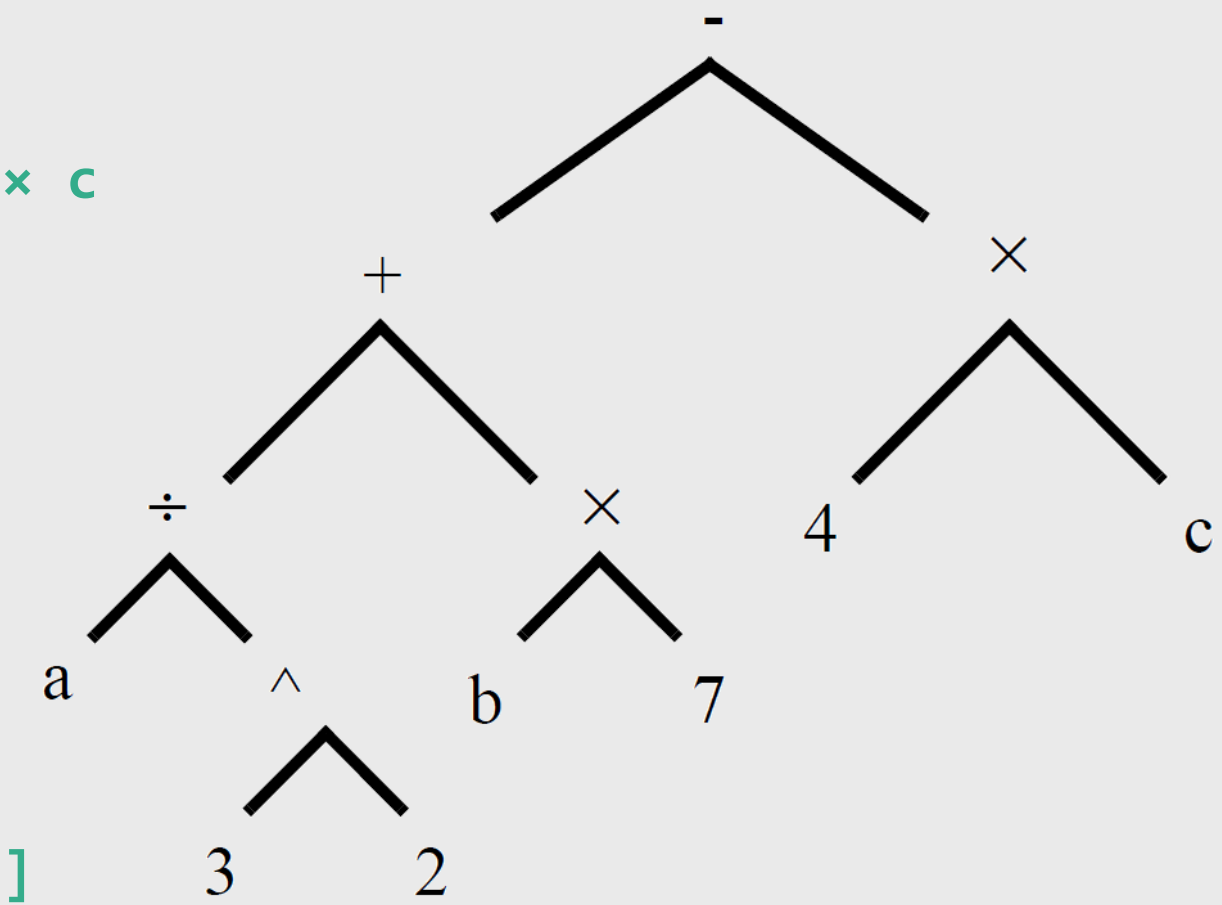
  x = a ÷ 3^2 + b × 7 - 4 × c

- Post-fix Arithmetic

  =

- Post-Order Tree Search

  [a 3 2^ ÷ b 7 × + 4 c × -]



Fundamental of Computer Science
CS1FC16: Lecture 05, Part – III

Exercises

Dr Varun Ojha

Department of Computer Science

University of Reading

Exercise

- Implement Single Linked List and show head prints first node data
- Write a program to implement stacks and show overflow and underflow error for push and pop operation.
- Trace expression tree for Pre-fix arithmetic, in-fix arithmetic, and arithmetic.