Go Track

PowerUp! SG Tech Traineeship – Software Engineering

Learning Objectives – Mod 9

At the end of the course, participants should be able to:

- Define what a data structure is.
- Examine the different data structures available.
- Demonstrate the different usage of each data structures discussed.

Data Structures

- Structures that come with programming language may not suit a problem.
- Therefore, sometimes it is necessary to create own data structures, that
  - Stores
  - Search
  - Retrieve data
- Sometimes data also needs to be manipulated or processed in specialized ways.
- Creating own data structures allows more control and flexibility to the kind of structures that can be constructed, and its supporting functions that it can have.

Data Structures

Types of Data Structures

- Linked lists
- Stacks
- Queues
- Binary Trees

Linked List

- A *Linked List* is a data structure consisting of a set of nodes, where each node consists of:
  - item (to store the data item)
  - next (to store a pointer that links to next node)
- The first node is usually accessed via a pointer in separate variable commonly called *head* (or *firstnode*).

Linked List

- Declaring a simple linked list node structure in Go.

```
type Node struct {
item itemType // to store the data item
next *Node // pointer to point to next node
}

type linkedList struct {
head *Node
size int
}
```

Linked List

- A new node can be created with its members initialized as follows:

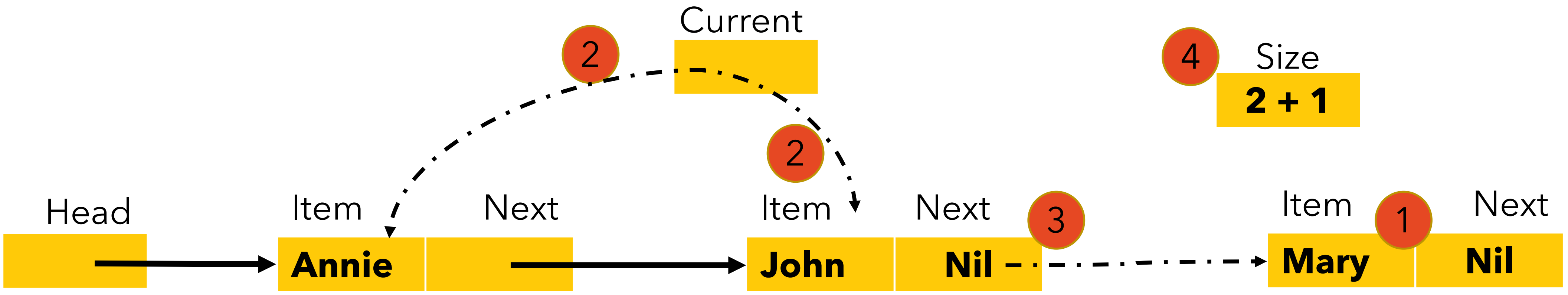
```
newNode := &Node{"Mary", nil}
```

or

```
newNode := &Node{
        item: "Mary",
        next: nil,
}
```

Linked List

- To add an item to the end of the list:
  - create a new node to store the item
  - traverse to the last node
  - make the last node's pointer to point to the new node
  - increase the size by 1

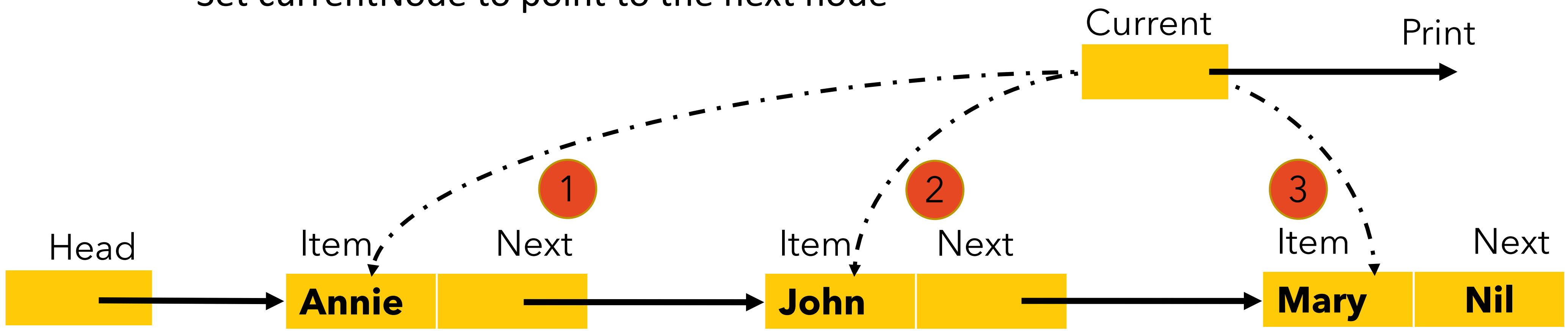


Linked List

```
func (p *linkedList) addNode(name string) error {
    newNode := &Node{
            item: name,
            next: nil,
    }
    if p.head == nil {
            p.head = newNode
    } else {
            currentNode := p.head
                    for currentNode.next != nil {
                    currentNode = currentNode.next
            }
            currentNode.next = newNode
    }
    p.size ++
    return nil
}
```

Linked List

- To display all the nodes in linked list:
  - Set a currentNode pointer to point to the first node of the list
  - While currentNode is not null
    - Retrieve the item from the node pointed by currentNode
    - Display the item
    - Set currentNode to point to the next node



Linked List

```
func (p *linkedList) printAllNodes() error {
    currentNode := p.head
    if currentNode == nil {
        fmt.Println("Linked list is empty.")
        return nil
    }
    fmt.Printf("%+v\n", currentNode.item)
    for currentNode.next != nil {
        currentNode = currentNode.next
        fmt.Printf("%+v\n", currentNode.item)
    }

    return nil
}
```

Linked List

- There can be many possible other functions for the linked list:
  - Adding item at a certain position
  - Removal of item from list
  - Retrieval of item at certain position from list
  - And many more…

Linked List

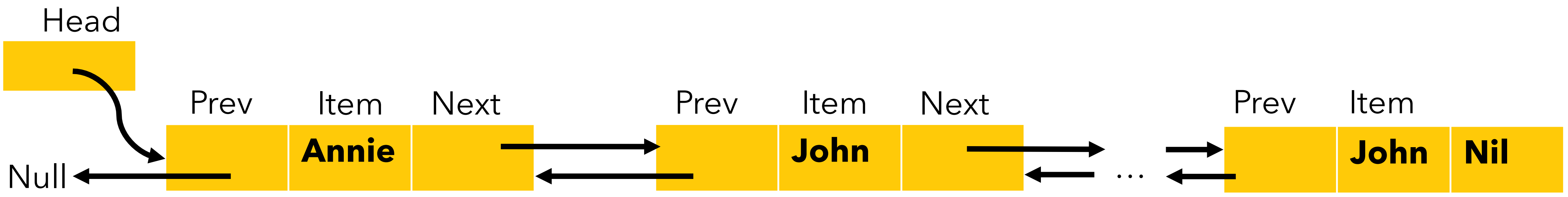
- Advantages of implementing Linked list:
  - Size of the list is not fixed, can grow as large as is necessary.
  - Easy to add/delete items by simply updating pointers, without shifting items but have to traverse to the correct location first.
  - Does not waste storage - use only necessary amount of memory (dynamic allocation)

Linked List

- Disadvantages of implementing Linked list:
  - access to items is sequential - need to traverse through the list sequentially to access an item
  - Worst case complexity? O(n) Adding item or removing the last item in the list.
  - Requires additional space to store the pointers/linkages

Doubly Linked List

- A *Doubly Linked List* is a data structure consisting of a set of nodes, where each node consists of:
  - item (to store the data item)
  - next (to store a pointer that links to next node)
  - prev (to store a pointer that links to the previous node)
- The first node is usually accessed via a pointer in separate variable commonly called *head* (or *firstnode*).
- The last node is commonly called *tail (or lastnode).*



Doubly Linked List

- Declaring a simple doubly linked list node structure in Go.

```
type Node struct {
        item itemType // to store the data item
        prev *Node   // pointer to point to prev node
        next *Node  // pointer to point to next node
}

type linkedList struct {
        head *Node
        tail *Node
        size int
}
```

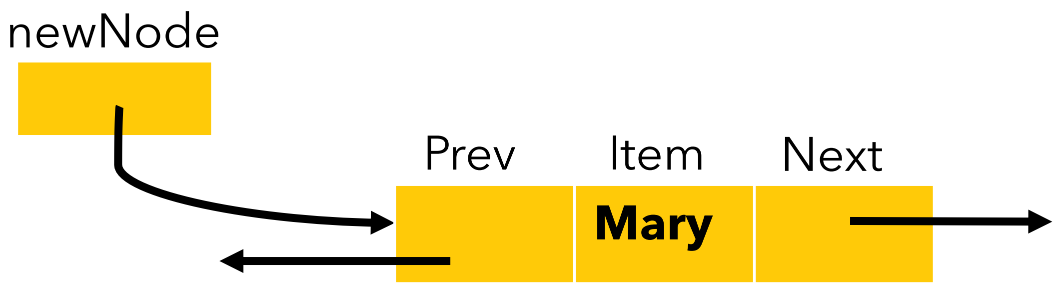
Doubly Linked List

- A new doubly node can be created with its members initialized as follows:

```
newNode := &Node{"Mary", nil, nil}
```

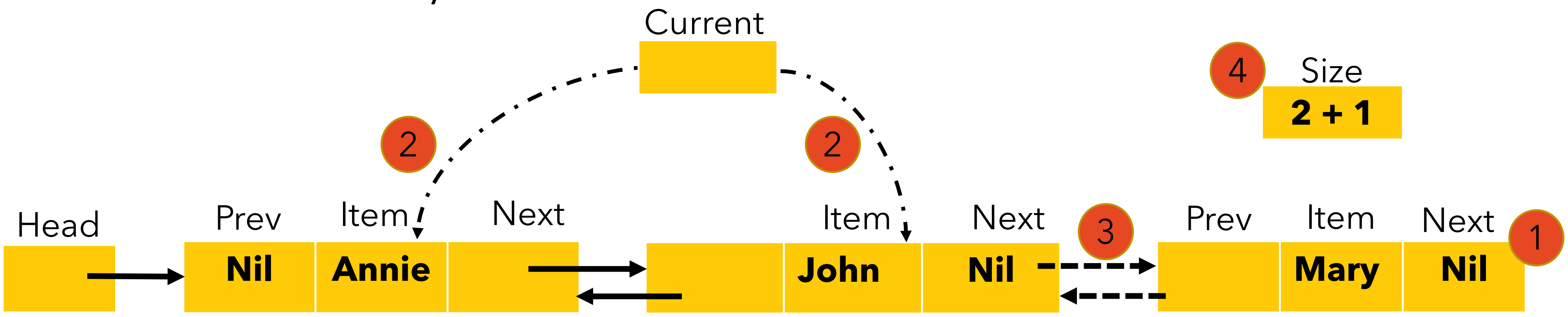
Or

```
newNode := &Node{
        item: "Mary",
        next: nil,
        prev: nil,
}
```



Doubly Linked List

- To add an item to the list:
  - create a new node to store the item
  - traverse to the selected node
  - make the last node's pointer to point to the new node
  - Make the new node's prev pointer to point to the last node
  - increase the size by 1

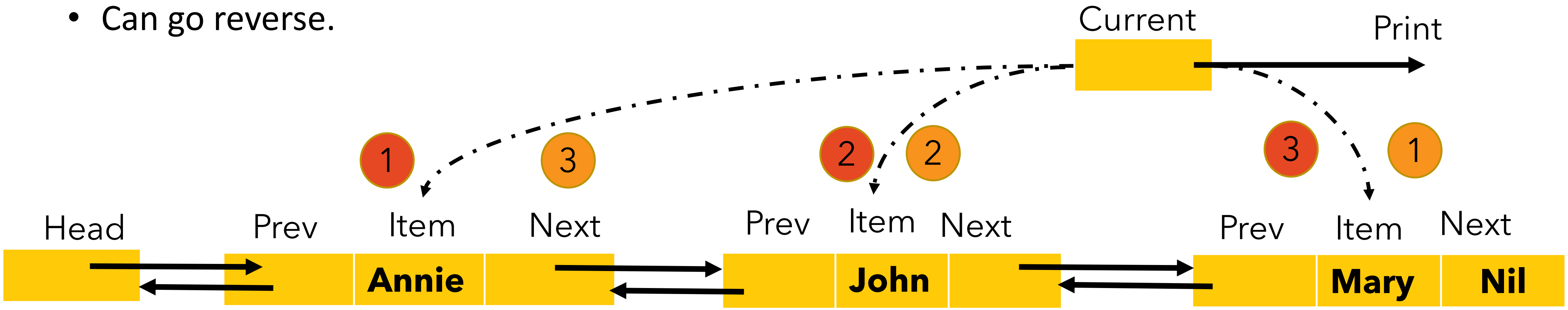


Doubly Linked List

```
func (p *linkedList) addNode(name string) error {
        newNode := &Node{
                item: name,
                next: nil,
                prev: nil,
        }
        if p.head == nil {
                p.head = newNode
                p.tail = newNode
        } else {
                currentNode := p.head
                for currentNode.next != nil {
                        currentNode = currentNode.next
                }
                newNode.prev = currentNode
                currentNode.next = newNode
                p.tail = newNode
        }
        p.size ++
        return nil
}
```

Doubly Linked List

- To display all the nodes in doubly linked list:
- Set a currentNode pointer to point to the first node of the list
- While currentNode is not null
  - Retrieve the item from the node pointed by currentNode
  - Display the item
  - Set currentNode to point to the next node
- Can go reverse.



Doubly Linked List

- To display in forward display

```
func (p *dblinklist) printAllNodes() error {
        currentNode := p.head
        if currentNode == nil {
                fmt.Println("DB Link list is empty")
                return nil
        }
        fmt.Printf("%+v\n", currentNode.item)
        for currentNode.next != nil {
                currentNode = currentNode.next
                fmt.Printf("%+v\n", currentNode.item)
        }
        return nil
}
```

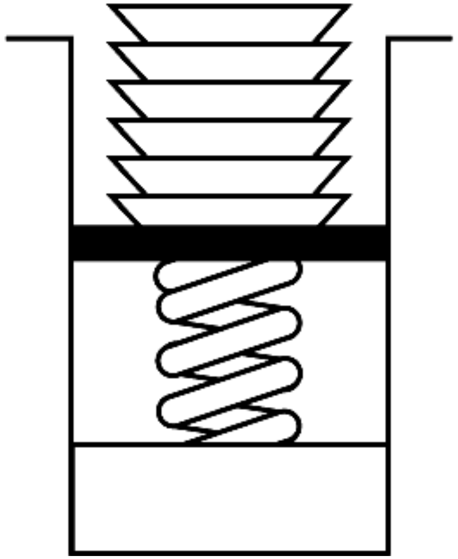
Doubly Linked List

- To display in reverse display

```
func (p *dblinklist) printAllNodesReverse() error {
	currentNode := p.tail
	if currentNode == nil {
		fmt.Println("DB Link list is empty")
		return nil
	}
	fmt.Printf("%+v\n", currentNode.item)
	for currentNode.prev != nil {
		fmt.Printf("%+v\n", currentNode.prev.item)
		currentNode = currentNode.prev
	}
	return nil
}
```

Stack

- A *Stack* is data structure for organizing data.
- Important property: *LIFO (Last-in First-Out)*
- Last item placed on the stack will be removed first.
- e.g. stack of plates, books etc
- only item at top of stack is visible and can be retrieved.
- new items can only be added to top of stack
- items can only be removed from top of stack
- Much simpler to implement than a linked list.

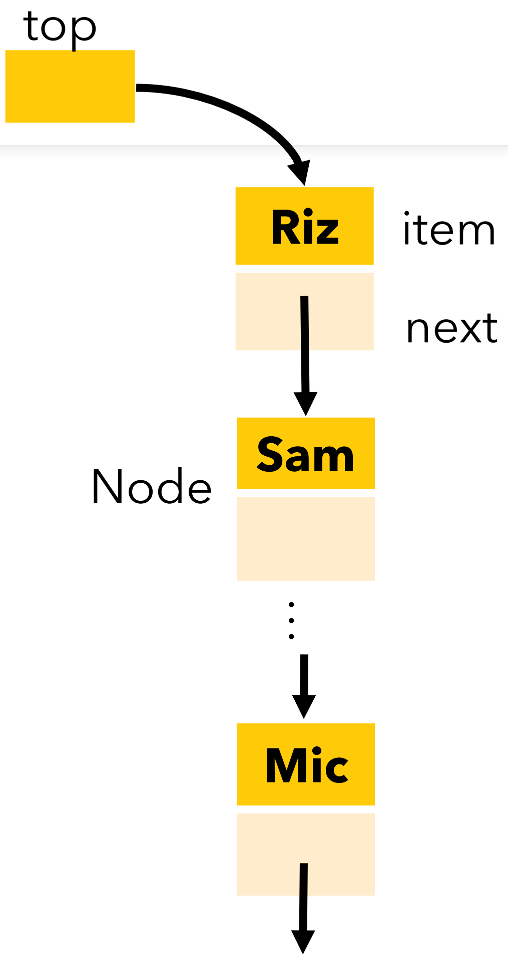


Stack

- Possible operations of a typical stack:
  - add an item to the top of stack (*push*)
  - remove an item from top of stack (*pop*)
  - retrieve/ look at item at top of stack (*peek* or *getTop*)

Stack

- A structure similar to linked list can be used to implement a stack.
- Similarly, it can consist of nodes, each of which contains:
  - item (to store the data item)
  - next (to store a pointer that links to next node)
- The first node is usually accessed via a pointer in separate variable called *top* (or *topNode*).



Stack

- A new node in stack is like a linked list, just rearranged.

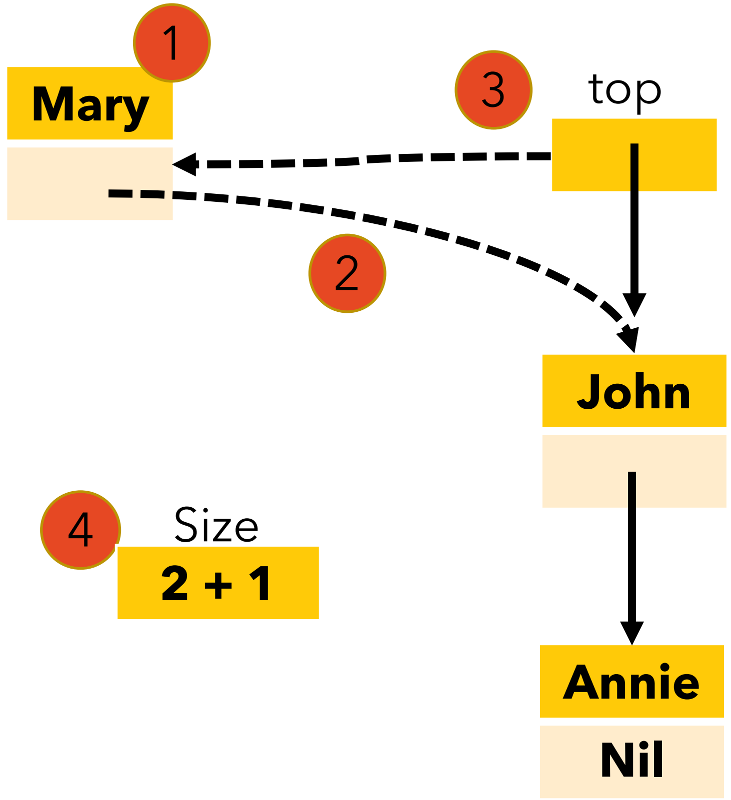
```
newNode := &Node{"Mary", nil}
```

Or

```
newNode := &Node{
item: "Mary", // to store the data item
next: nil, // to store the data item
}
```

Stack

- To *push* a new item on top of a stack:
  - create a new node to store the item
  - make the new node's next point to the top
  - make the top point to new node
  - increase the size by 1

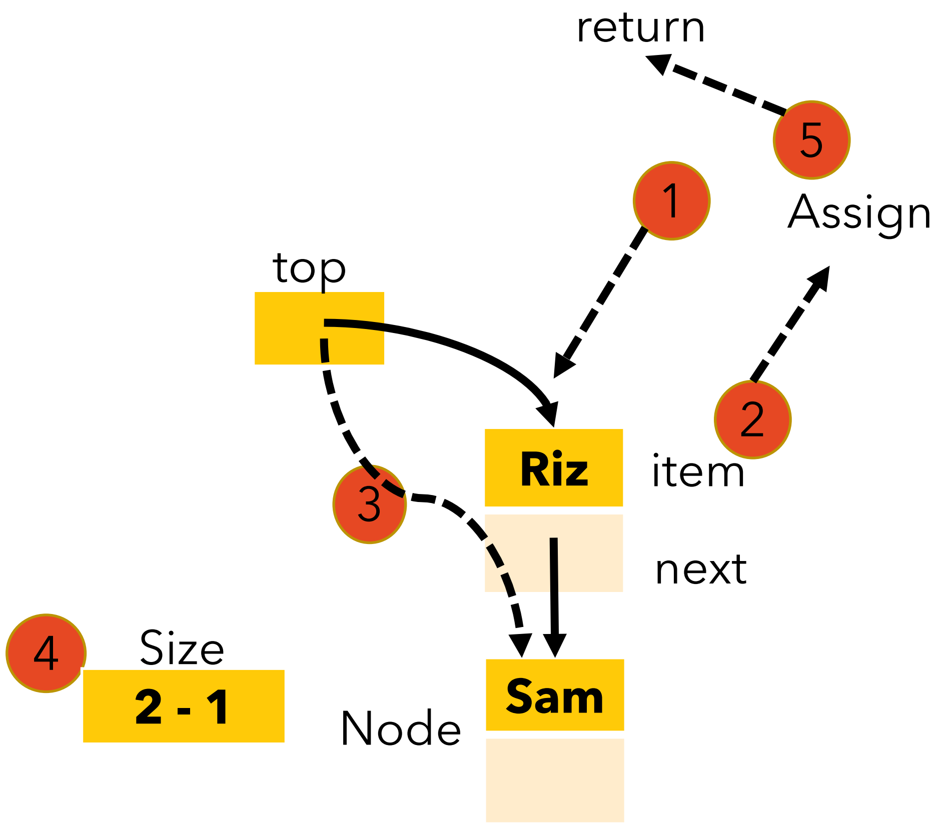


Stack

```
func (p *stack) push(name string) error {
    newNode := &Node{
        item: name,
        next: nil,
    }
    if p.top == nil {
        p.top = newNode
    } else {
        newNode.next = p.top
        p.top = newNode
    }
    p.size ++
    return nil
}
```

Stack

- To pop an item from top of a stack:
  - Need to check if top is pointing to any node or nil
  - Otherwise, capture the item at top node.
  - Need to check if size is 1, make top point to nil
  - Else, make the top point to the next node in stack
  - Decrease the size by 1
  - Return the popped item.



Stack

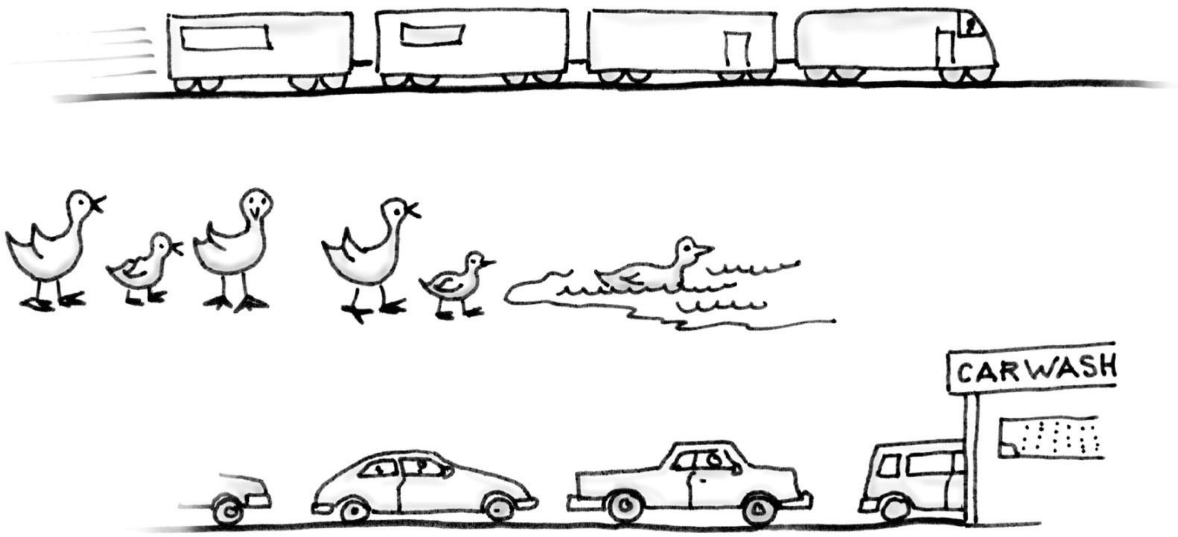
```
func (p *stack) pop() (string, error) {
    var item string
    if p.top == nil {
        return "", errors.New("Empty Stack!")
    }
    item = p.top.item

    if p.size == 1 {
        p.top = nil
    } else {
        p.top = p.top.next
    }
    p.size--
    return item, nil
}
```

Stack

- Stacks are used in several applications:
  - Checking for balanced parenthesis in compilers
  - Converting infix to postfix expressions
  - Evaluation of postfix expressions
  - Simulation of applications that has LIFO behaviour
  - And so on…

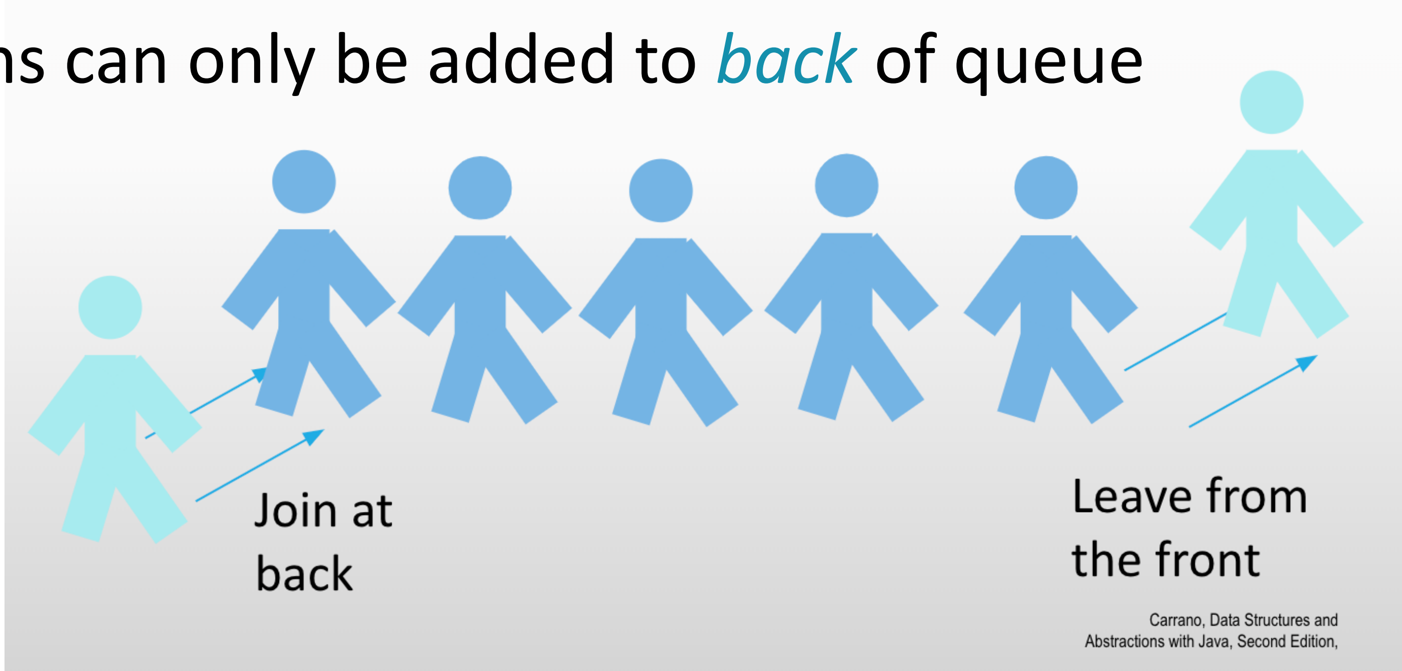
Queue

- A *queue* is another form of data structure for organizing data.
- Important property: *FIFO (First-in First-Out)*
- First item placed on the queue will be removed first.



Queue

- operations can only occur at the queue's two ends
- Items can only be removed from *front* of queue
- New items can only be added to *back* of queue

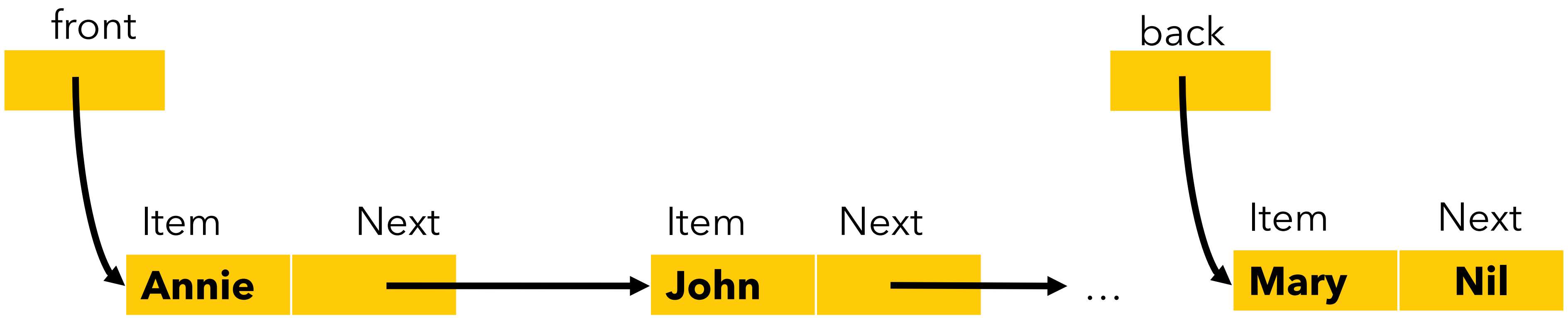


Queue

- Possible operations of a typical queue:
  - add an item to back of queue(*enqueue*)
  - remove an item from front of queue (*dequeue*)
  - retrieve/ look at item at front of stack (*getFront*)

Queue

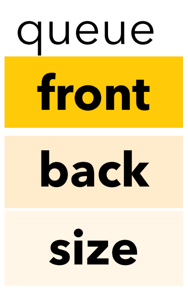
- A structure similar to linked list can be used to implement a queue.
- Similarly, it can consist of nodes, each of which contains:
    - item (to store the data item)
    - next (to store a pointer that links to next node)
- a pointer *front* (or *frontNode*) to point to the node at front position.
- a pointer *back* (or *backNode*) to point to the node at back position.



Queue

- Format for declaring a queue structure.

```
type Node struct {
       item itemType // to store the data item
       next *Node  // pointer to point to next node
}

type queue struct {
       front *Node
       back *Node
       size int
}
```



Queue

- Recall the creation of node

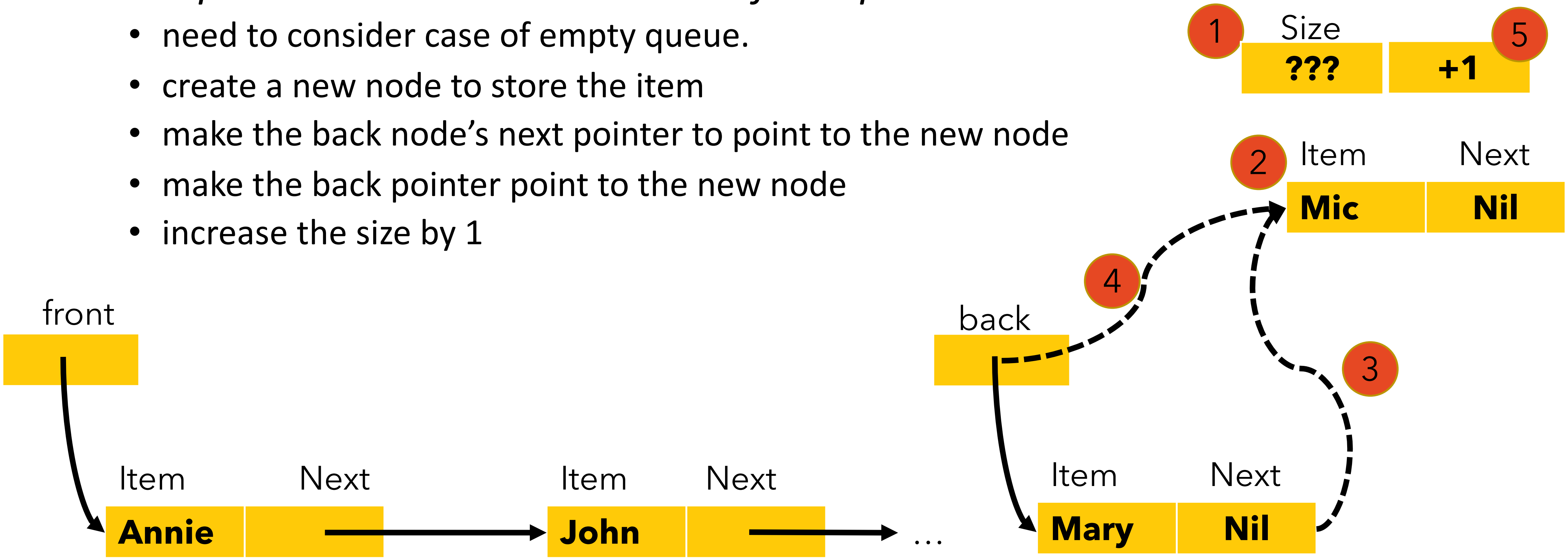
```
newNode := &Node{"Mary", nil}
```

Or

```
newNode := &Node{
item: "Mary", // to store the data item
next: nil, // to store the data item
}
```

Queue

- To *enqueue a new item at the back of the queue*:
  - need to consider case of empty queue.
  - create a new node to store the item
  - make the back node's next pointer to point to the new node
  - make the back pointer point to the new node
  - increase the size by 1

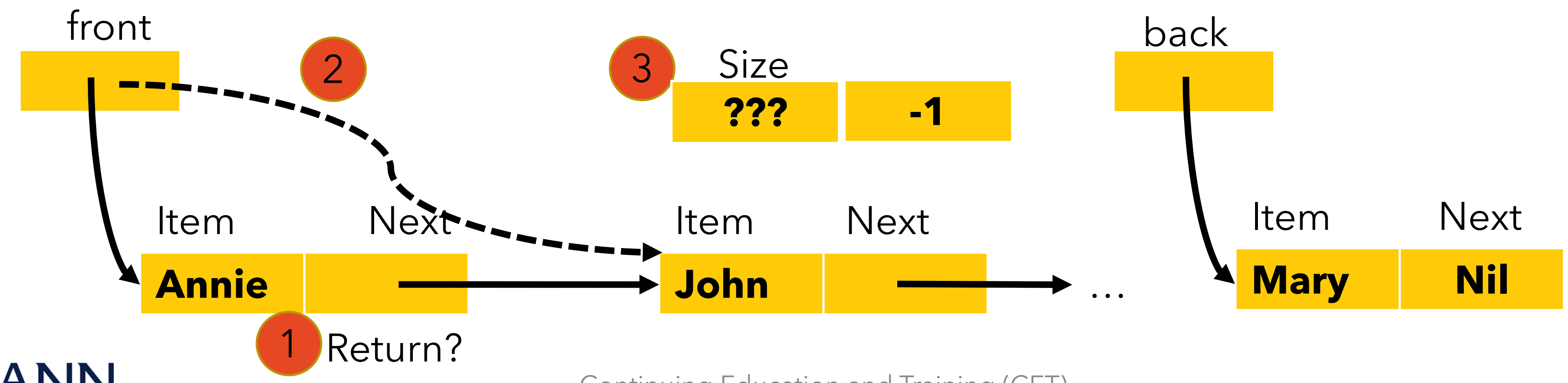


Queue

```
func (p *queue) enqueue(name string) error {
	newNode := &Node{
		item: name,
		next: nil,
	}
	if p.front == nil {
		p.front = newNode
	} else {
		p.back.next = newNode

	}
	p.back = newNode
	p.size++
	return nil
}
```

Queue

- To dequeue an item from front of a queue:
  - Capture the item of the node at front
  - Make the front point to the next node in queue
  - Return the item if needed.
  - Reduce size by 1
  - Need to consider case of only 1 node.



Queue

```
func (p *queue) dequeue() (string, error) {
        var item string

        if p.front == nil {
                return "", errors.New("Empty Queue!")
        }

        item = p.front.item
        if p.size == 1 {
                p.front = nil
                p.back = nil
        } else {
                p.front = p.front.next
        }
        p.size--
        return item, nil
}
```

Queue

- Advantages of implementing pointer-based Queue:
  - Size of the list is not fixed - can grow as large as or shrink as is necessary
  - Adding & removing data involves manipulation of linkages including front and back , O(1) time complexity
  - Does not waste storage - use only necessary amount of memory (dynamic allocation)

- Disadvantages of implementing pointer-based Queue:
  - Additional memory needed to store the linkages.

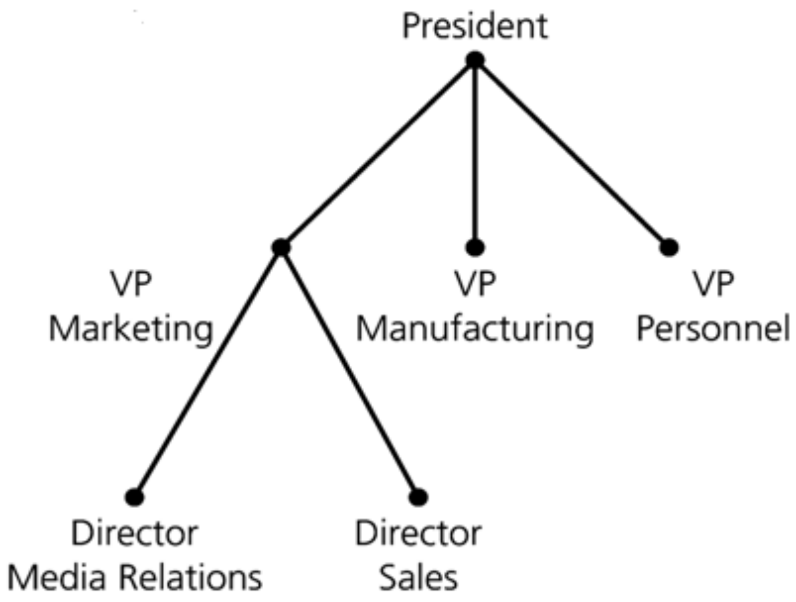
Queue

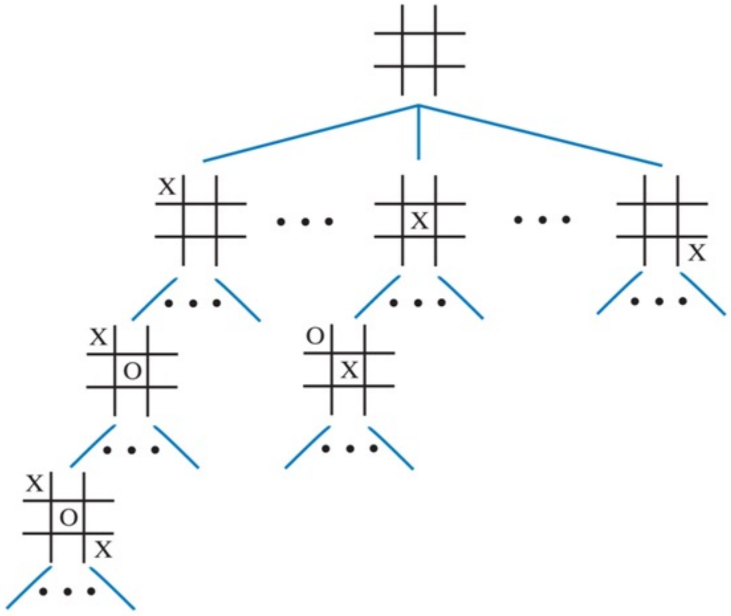
- Queues are used in several applications:
  - Processing jobs that are added in and cleared in FIFO manner e.g. CPU scheduling, disk scheduling, enterprise resource planning, customer service management, order handling, call centres, queue management etc etc
  - Implementing buffers to hold data that are transferred asynchronously (receiving not at same rate as sending) e.g. IO buffers, file IO

- And many many more…

Tree

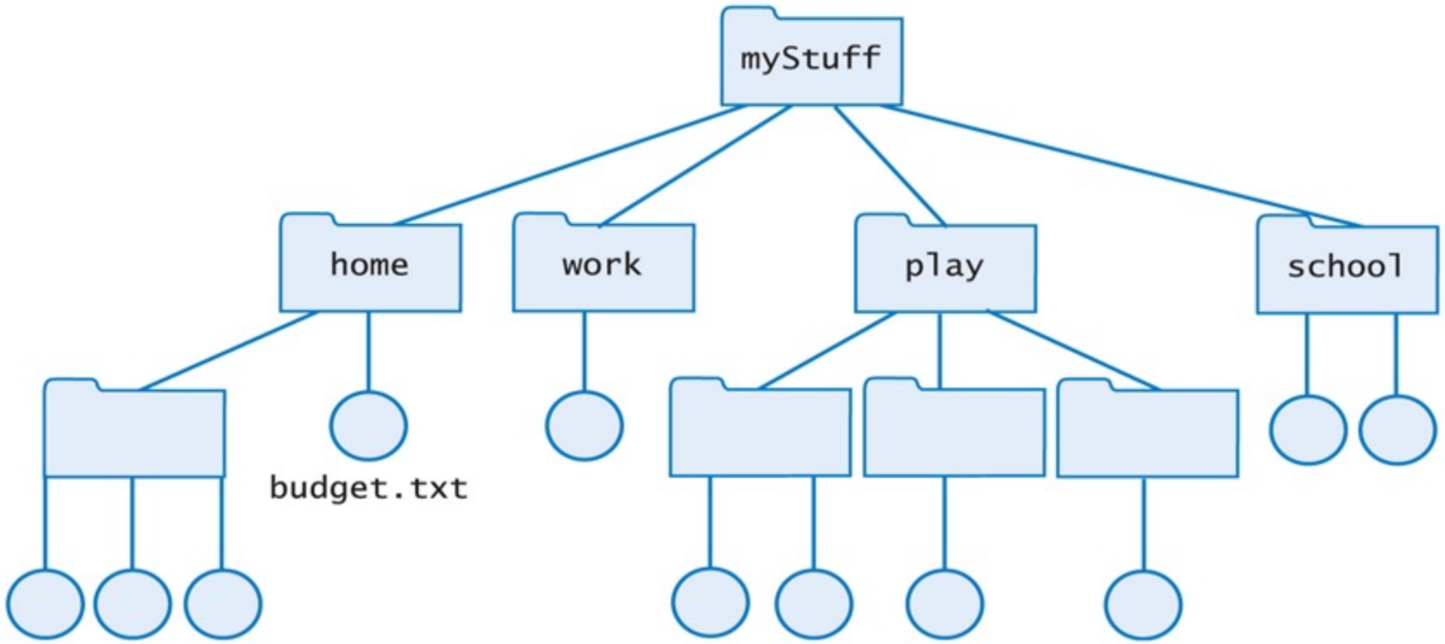
- A *tree* is a data structure that organizes data in hierarchical order.
  - e.g. organizational chart , game tree, family tree, file directories, expression trees, decision trees





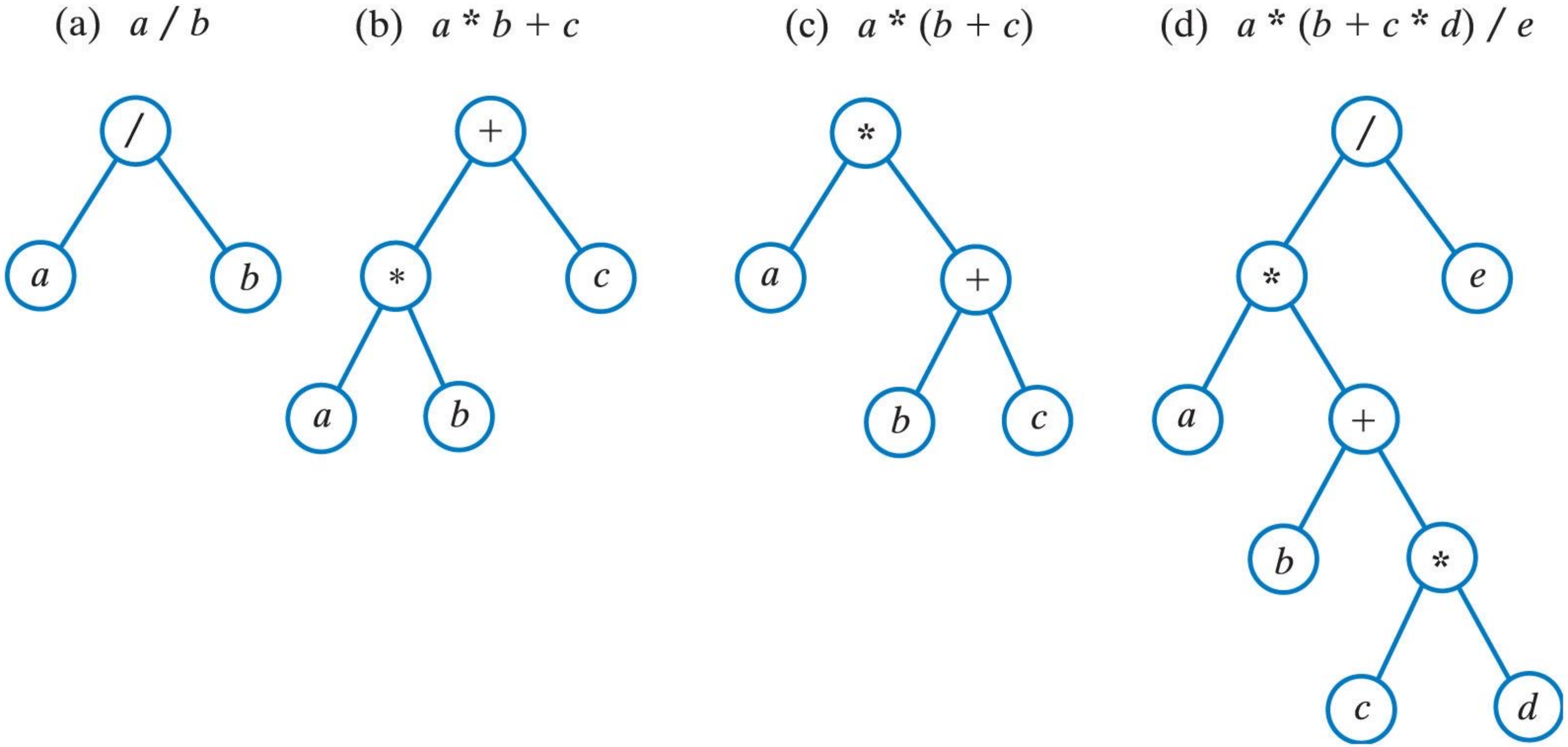
Tree

- Another example: directory structure…



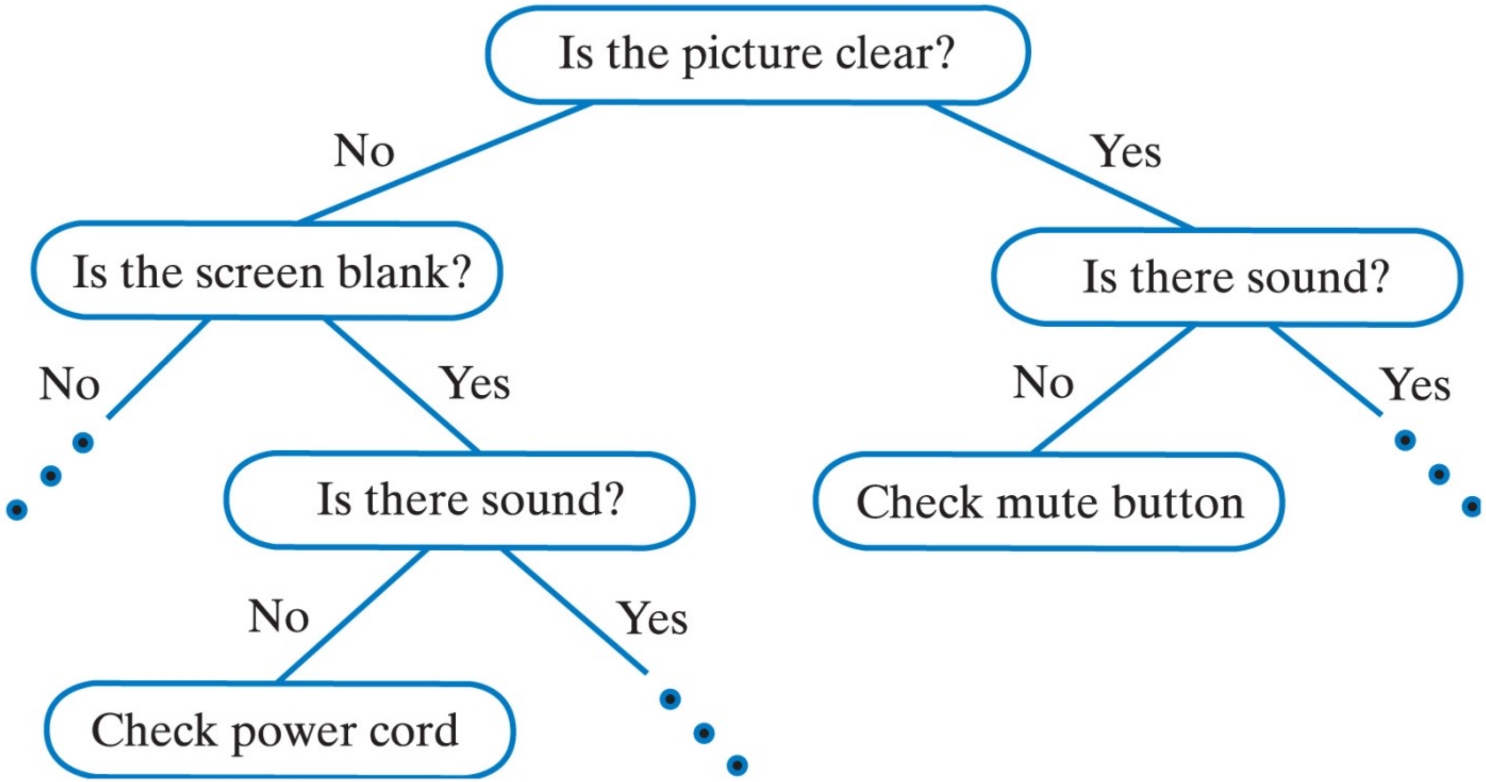
Tree

- Expression trees:



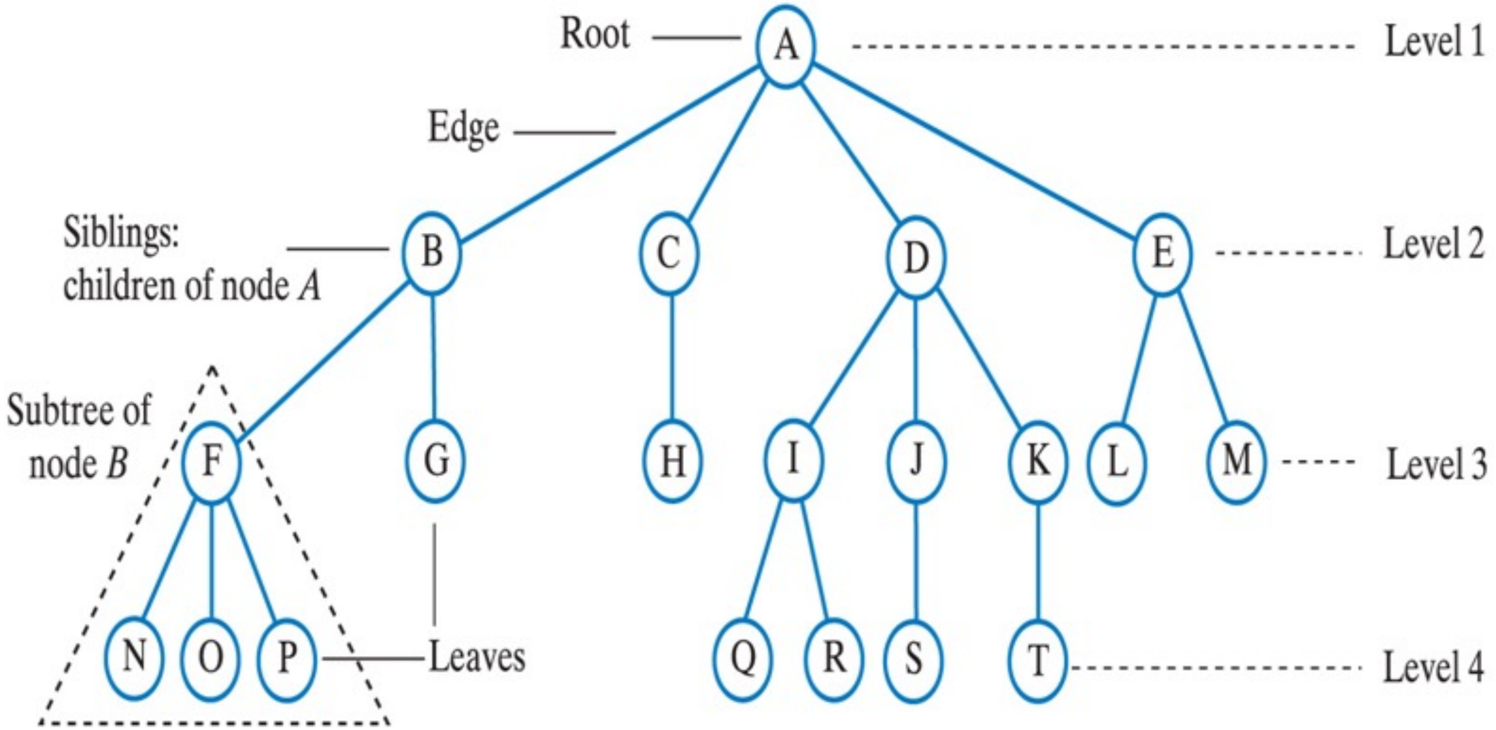
Tree

- Decision trees:



Tree

- Tree Terminology:
  - Trees are composed of *nodes* and *edges*



Tree

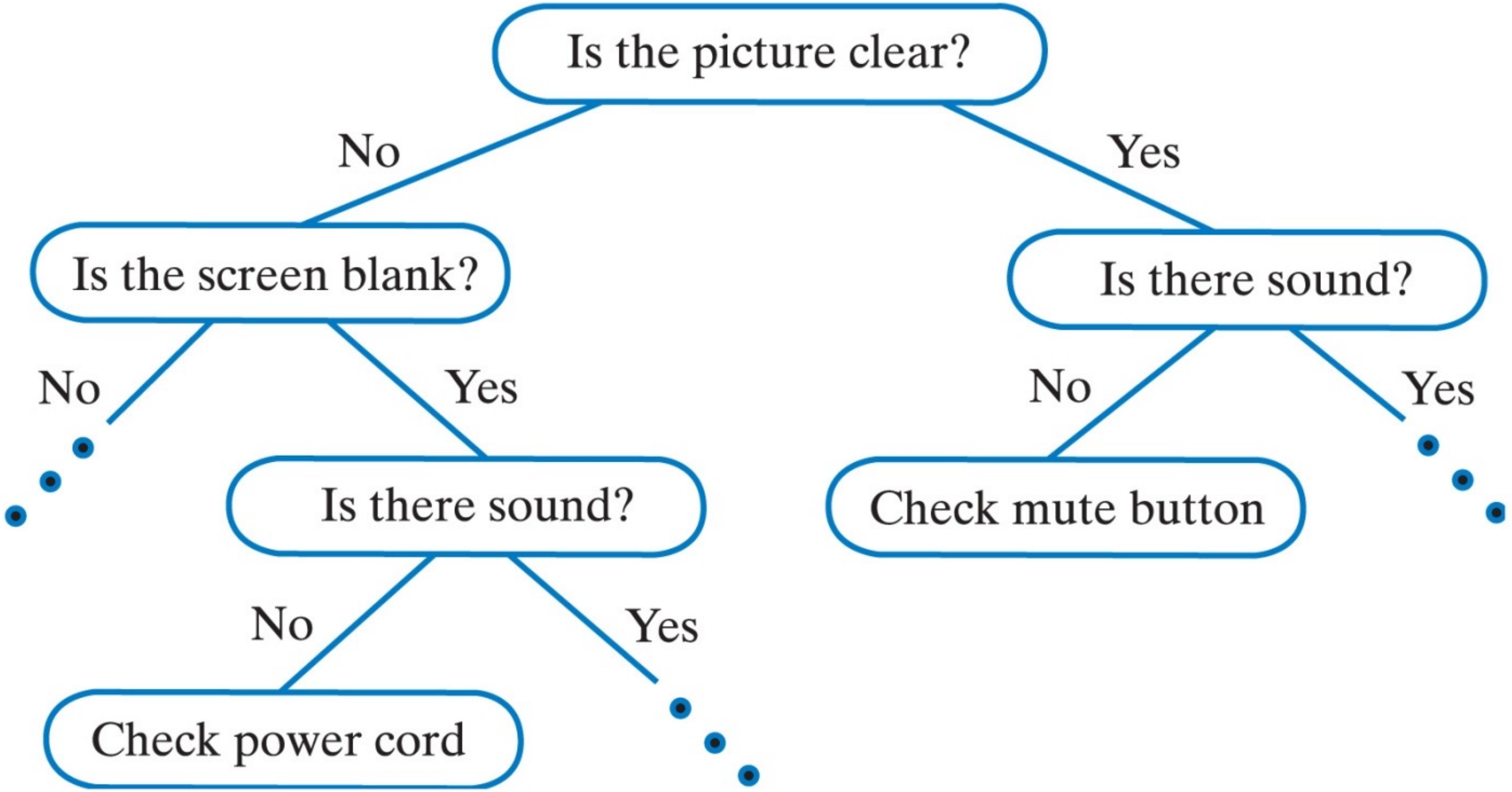
- *root* - the node at the top of the tree
- *leaf* - a node with no children
- *level (or height/ depth)* - the number of generations from the root , longest path from root to leaf
    - not to be confused with depth of a node - path from root to the node
- *height* - the number of levels in the tree
- *parent* - node with nodes (children) below it
- *children* - nodes below a given node (parent)

Tree

- Types of Trees
  - *General Tree* - a tree with any number of subtrees
  - *Binary Tree* - a tree with at most 2 subtrees
  - *Binary Search Tree* - a binary tree that is ordered
    - values on the left subtree < value of parent
    - values on the right subtree > value of parent
  - *AVL Tree* - a binary search tree that is balanced
    - the heights of any node's two subtrees differ by at most 1

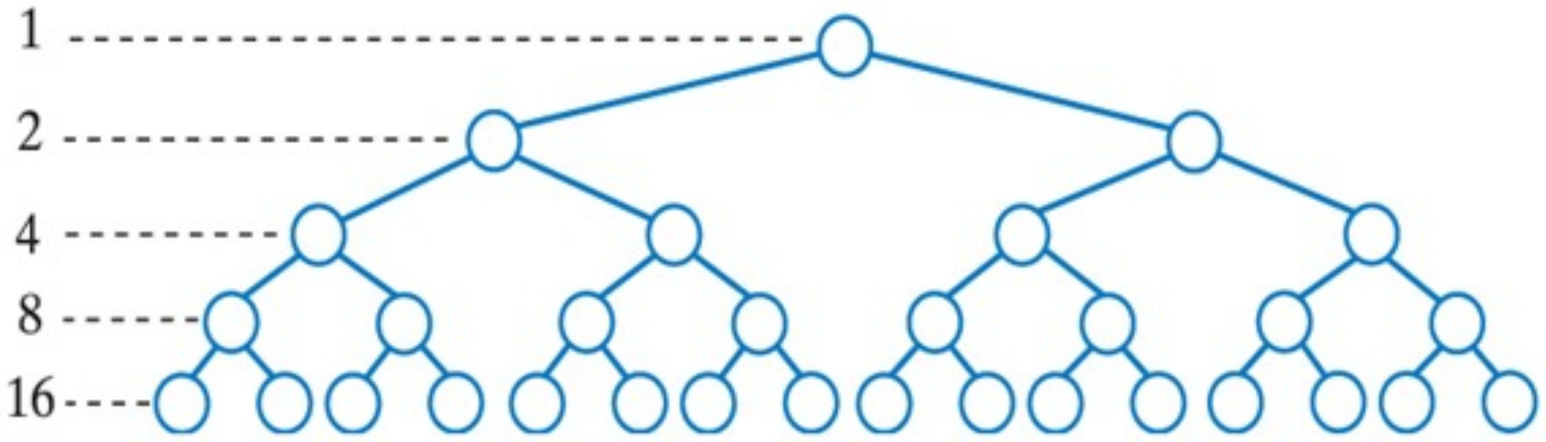
Binary Tree

- A *binary tree* is a tree that has at most 2 *subtrees.*



Binary Tree

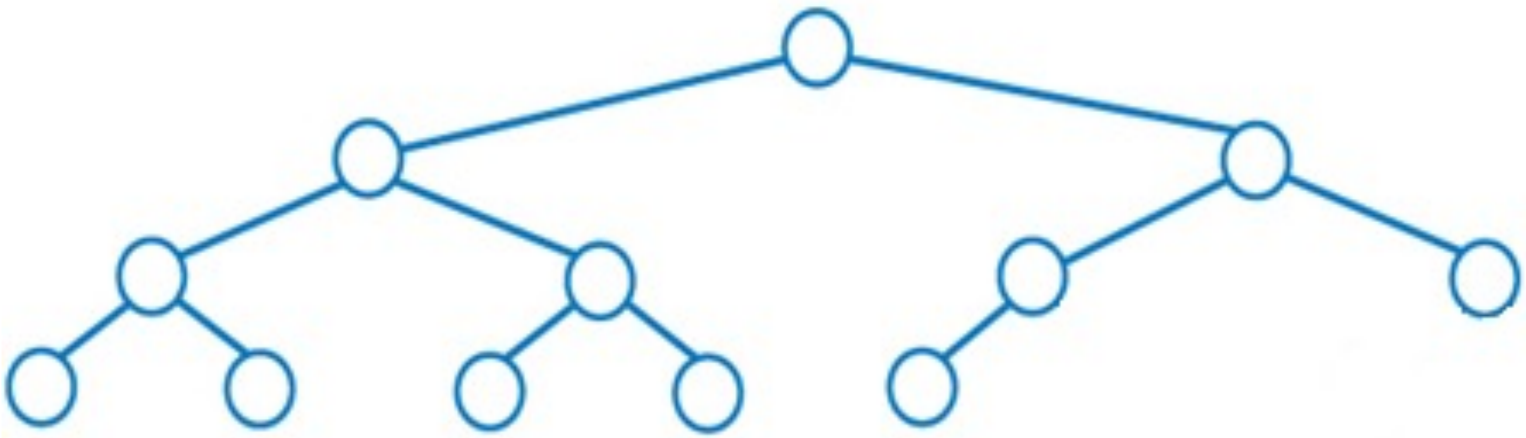
- A binary tree is *full* if every node (except the leaf nodes) has two children



The number of nodes in a full binary tree of height h, **n = 2h – 1**
The height of a full binary tree with n nodes that is full is **h = log2(n + 1)**

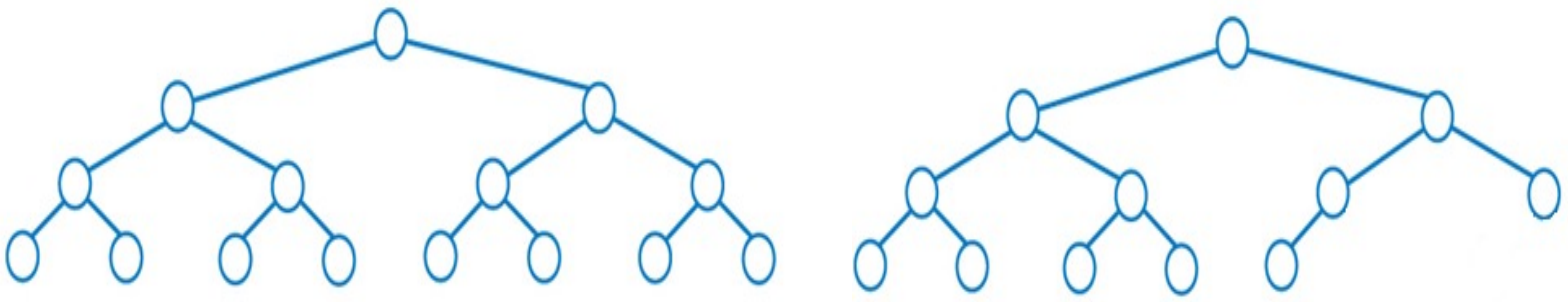
Binary Tree

- A binary tree is *complete* if
  - it is full to all the levels except the last level and
  - the last level is filled from left to right



Binary Tree

- A binary tree is *balanced* if the heights of any node's two subtrees differ by at most 1



Binary Tree

- Traversals of a Binary Tree

  Inorder : Left-Root-Right

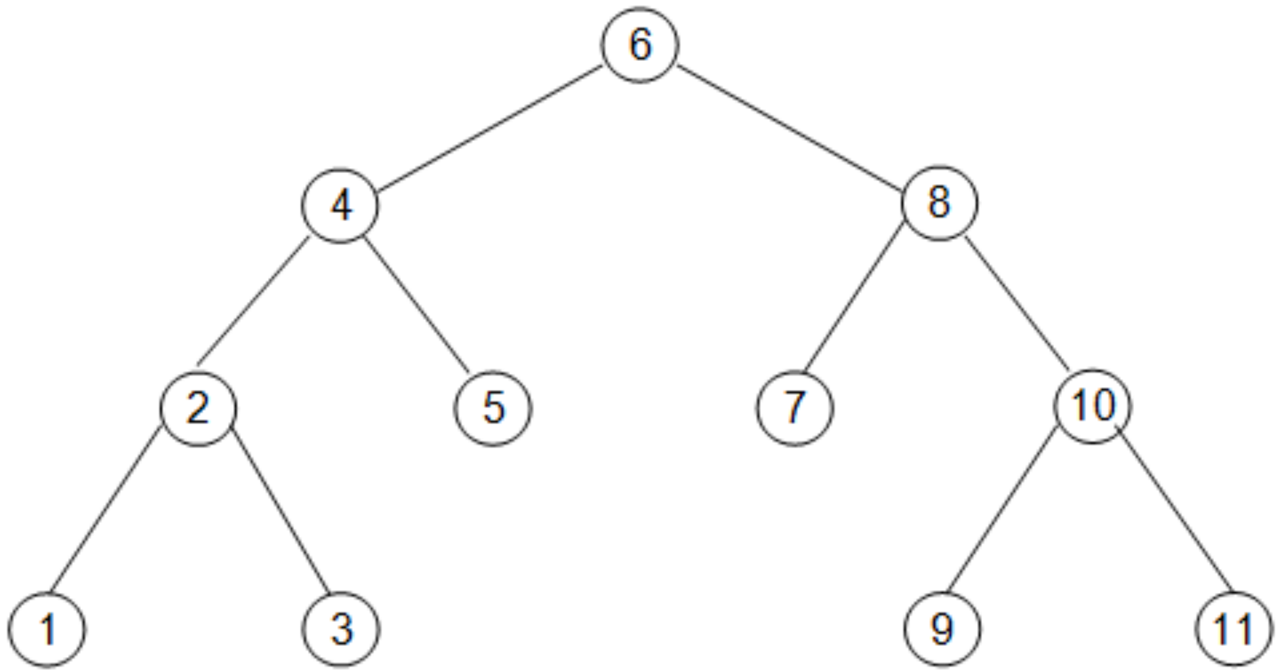
  Preorder : Root-Left-Right

  Postorder : Left-Right-Root

  Level order : Level by Level

Binary Tree

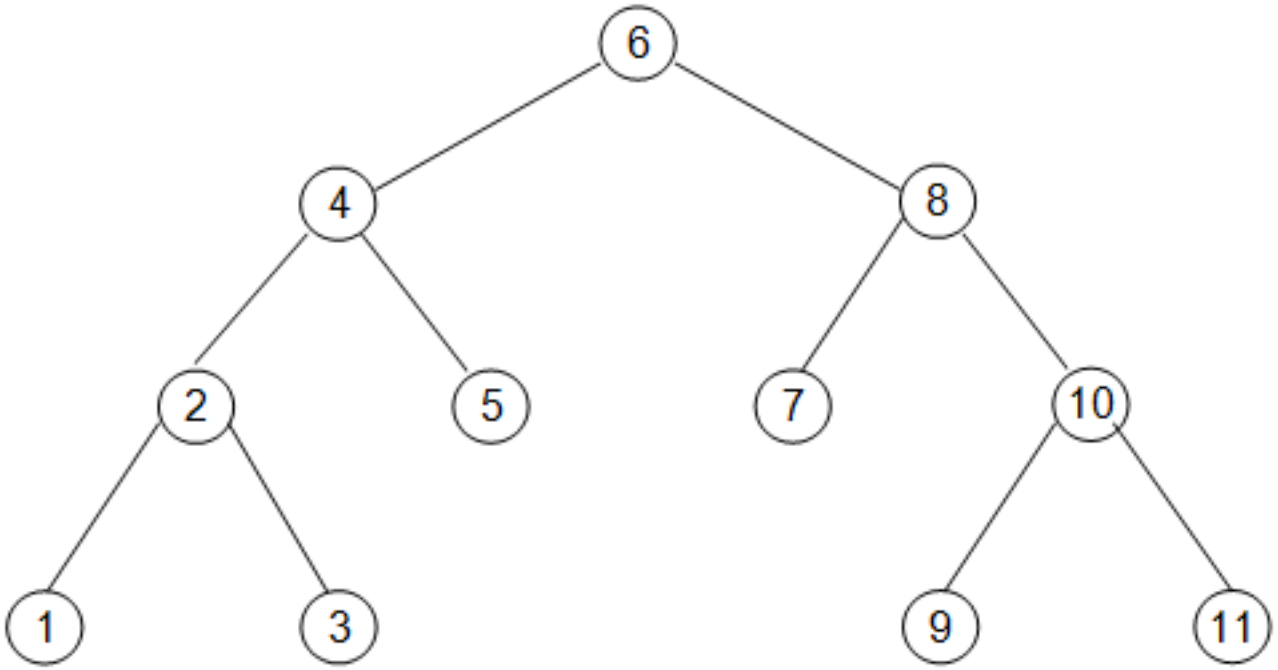
- *Inorder Traversal* : Left-Root-Right



1 2 3 4 5 6 7 8 9 10 11

Binary Tree

- *Preorder Traversal* : Root-Left-Right

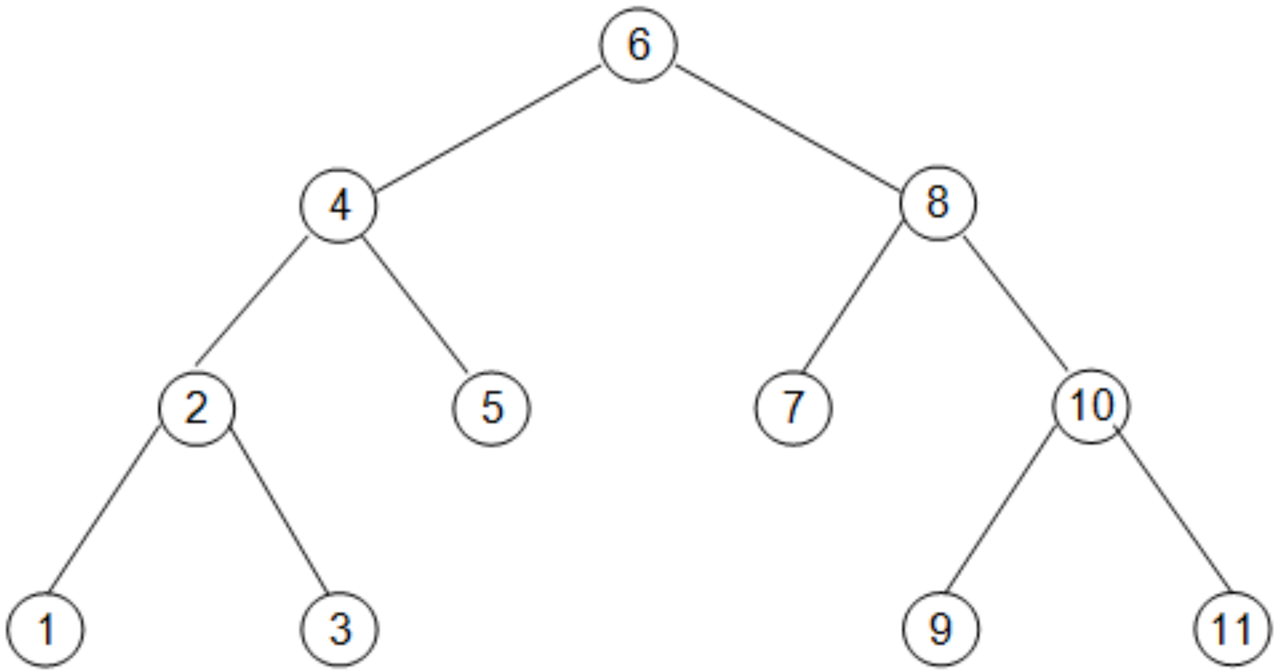


6 4 2 1 3 5 8 7 10 9 11

NGEE ANN SCHOOL OF INFOCOMM TECHNOLOGY

Binary Tree

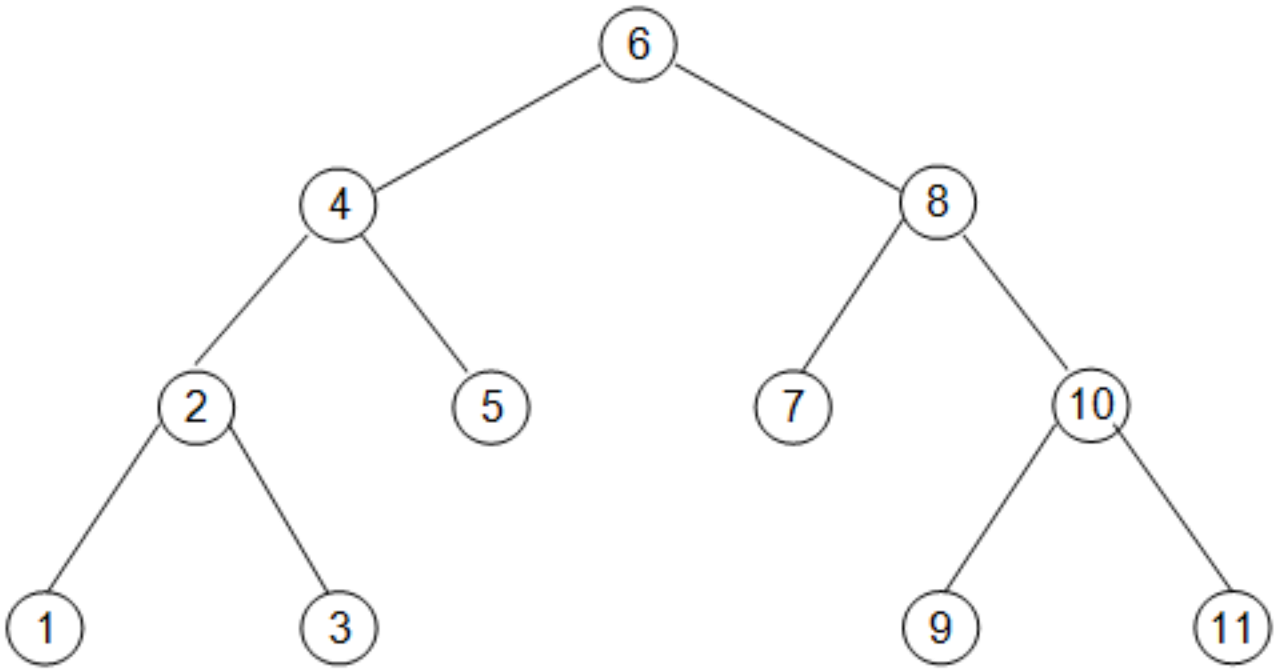
- *Postorder Traversal* : Left-Right-Root



1 3 2 5 4 7 9 11 10 8 6

Binary Tree

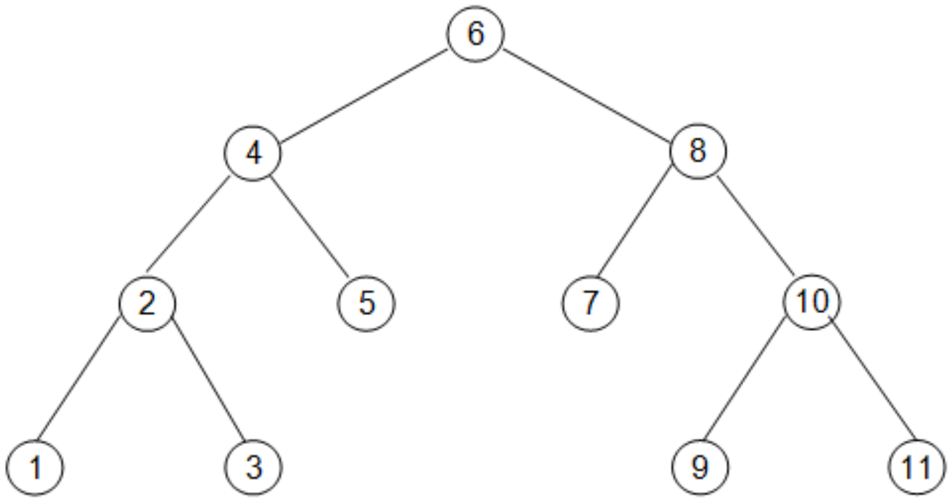
- *Level order Traversal* : Level by level

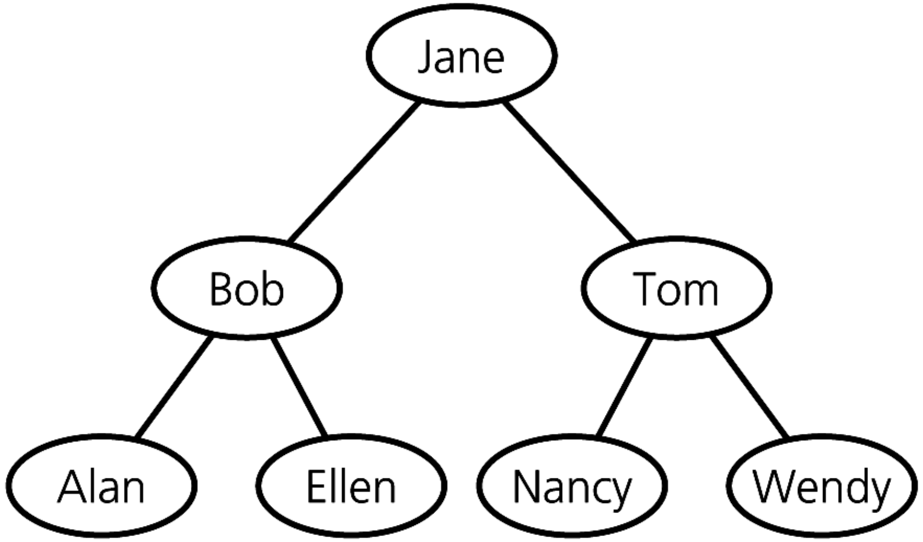


6 4 8 2 5 7 10 1 3 9 11

Binary Search Tree

- A binary search tree is a binary tree that is ordered
  - values in the left subtree < value of parent
  - values in the right subtree > value of parent



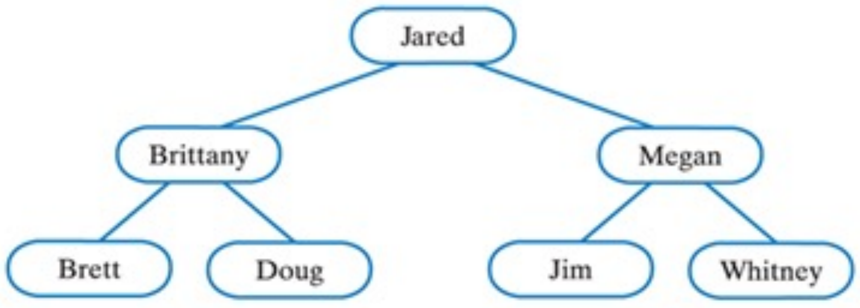


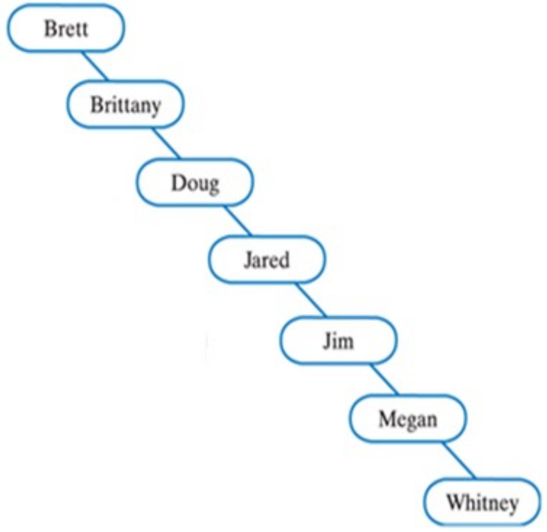
Binary Search Tree

- All the operations in a BST require a search that begins at the root
- Number of comparisons is directly proportional to the height of the tree.
- Worst case (balanced) = $O(\log_2 n)$
- Worst case (unbalanced) = $O(n)$, where n = number of nodes in the binary search tree
- Therefore, it is important that a binary search tree is *balanced*
- A binary search tree usually becomes unbalanced during an insert or removal of item.
- So how to make it balanced? Find out more about AVL trees!

Binary Search Tree

- The speed of finding the same element say Megan would be different in a balanced and an unbalanced tree.





Binary Search Tree

- Possible operations of a binary search tree:
  - Insert
  - Search
  - Traversal
  - Remove

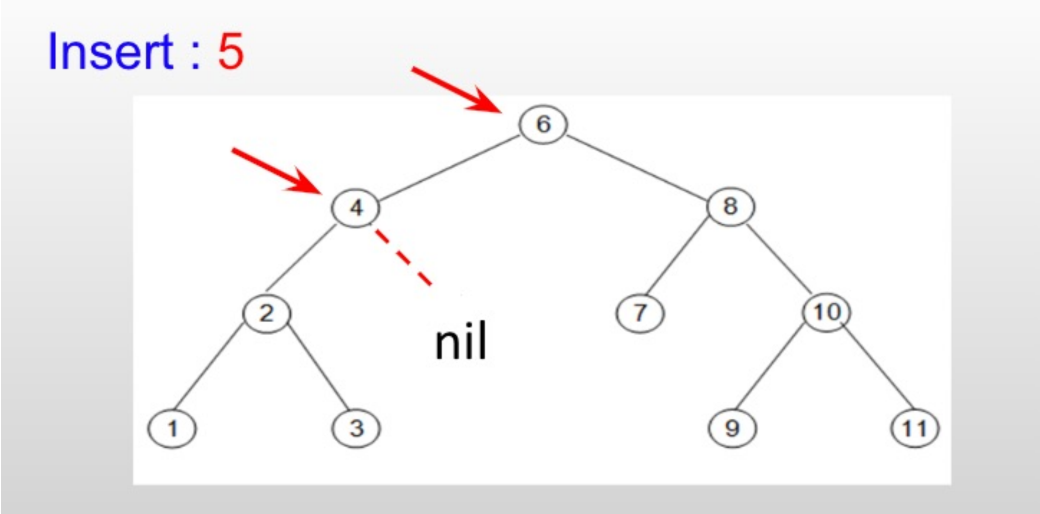
Binary Search Tree

- The format for declaring the tree structure:

```
type BinaryNode struct {
    item itemType // to store the data item
    left *BinaryNode // pointer to point to left node
    right *BinaryNode // pointer to point to right node
}

type BST struct {
    root *BinaryNode
}
```

Binary Search Tree

- To *insert a new item into the BST*:
  - Search for the item (pointer will point to nil)
  - Create a new node to store the item
  - Set the pointer pointing to null to point to new node



Binary Search Tree

```
func (bst *BST) insertNode(t **BinaryNode, item string) error {

    if *t == nil {
        newNode := &BinaryNode{
            item:  item,
            left:  nil,
            right: nil,
        }
        *t = newNode
        return nil
    }

    if item < (*t).item {
        bst.insertNode(&((*t).left), item)
    } else {
        bst.insertNode(&((*t).right), item)
    }
    return nil
}
```

Binary Search Tree

```
func (bst *BST) insert(item string) {
    bst.insertNode(&bst.root, item)

}
```

Binary Search Tree

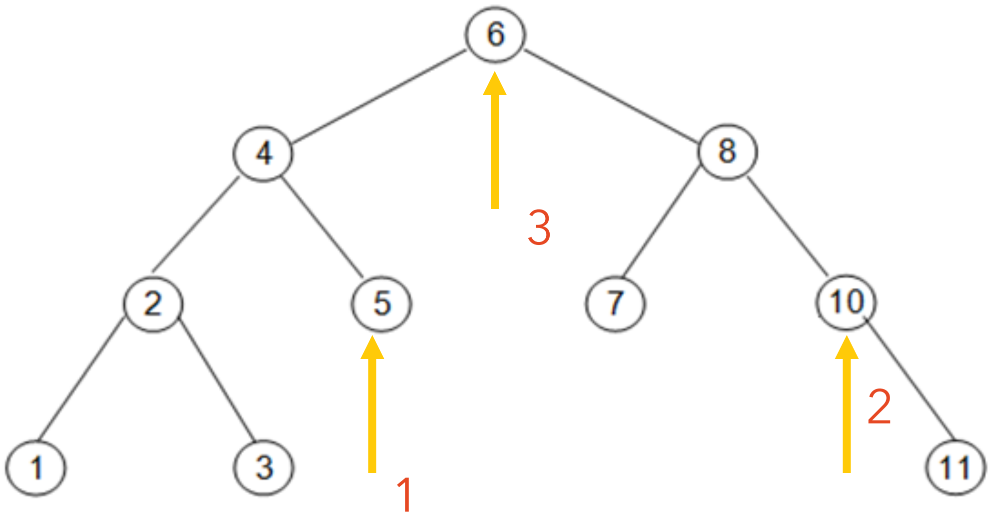
- To *traverse a binary tree in InOrder manner*:
  - Check if the current node is nil
  - If not, recursively call inOrder traversal on left subtree
  - Upon returning, process the current node
  - Then recursively call inOrder traversal on right subtree

Binary Search Tree

```go
func (bst *BST) inOrderTraverse(t *BinaryNode) {

    if t != nil {
        bst.inOrderTraverse(t.left)
        fmt.Println(t.item)
        bst.inOrderTraverse(t.right)
    }
}

func (bst *BST) inOrder() {
    bst.inOrderTraverse(bst.root)
}
```

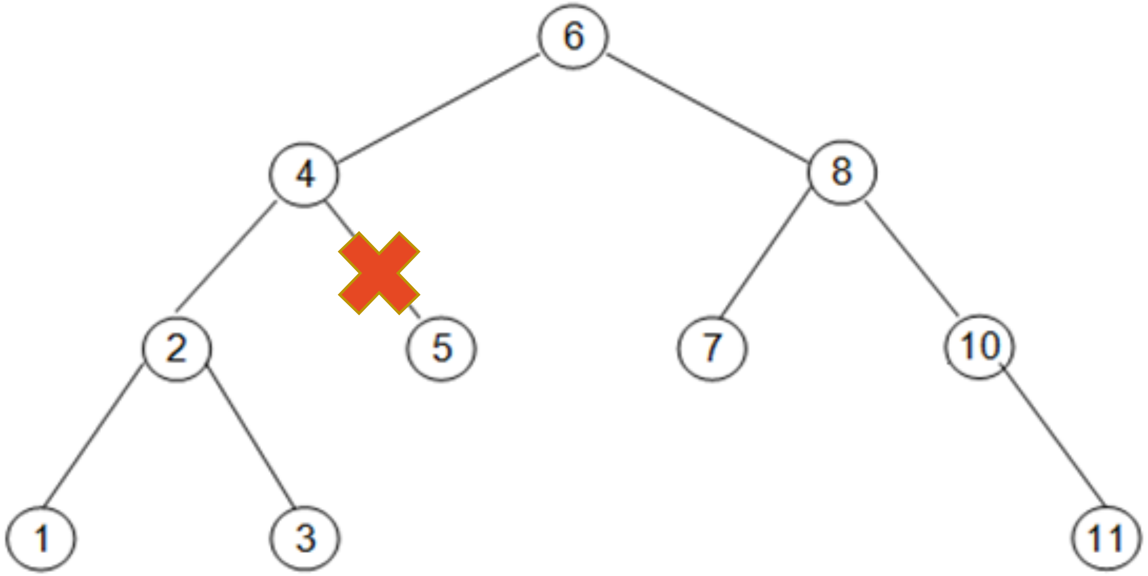
Binary Search Tree

- To remove a new item into the BST:
  - 3 Distinct Possible Cases
    - Case 1 : node to be deleted has 0 child (is a leaf)
    - Case 2 : node to be deleted has 1 child
    - Case 3 : node to be deleted has 2 children



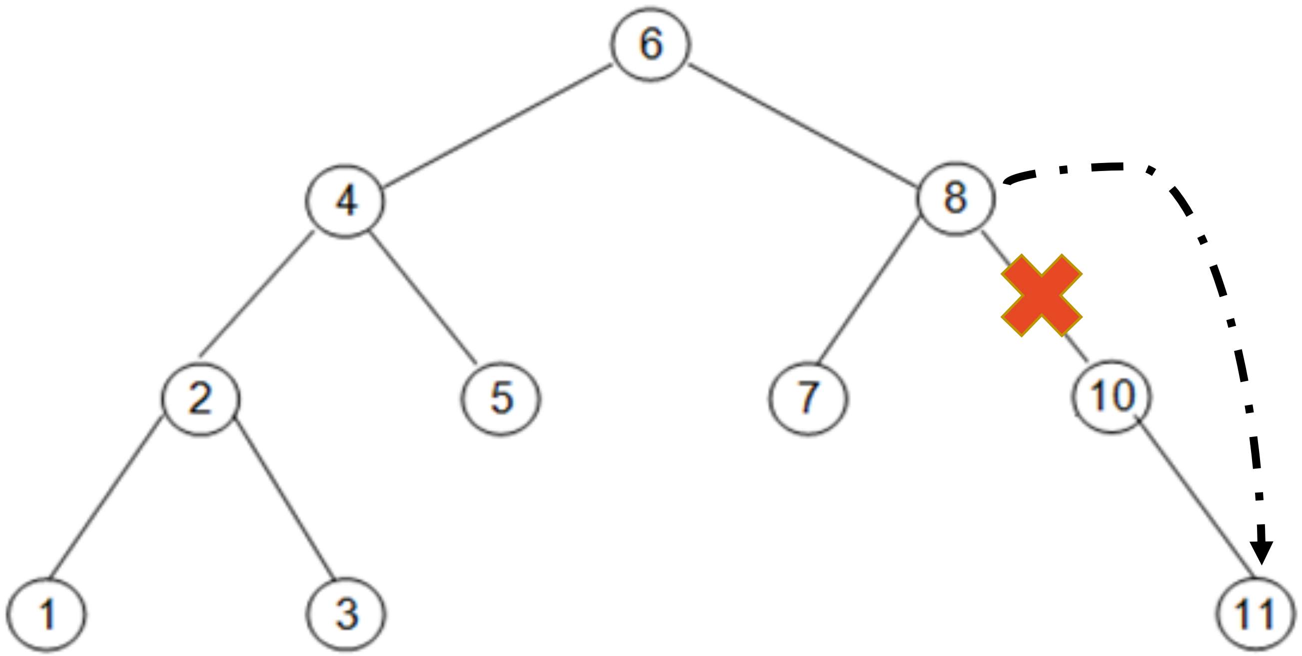
Binary Search Tree

- Case 1: Deleting a leaf node
  - Simply delete the node by setting pointer pointing to it to point to nil.



Binary Search Tree

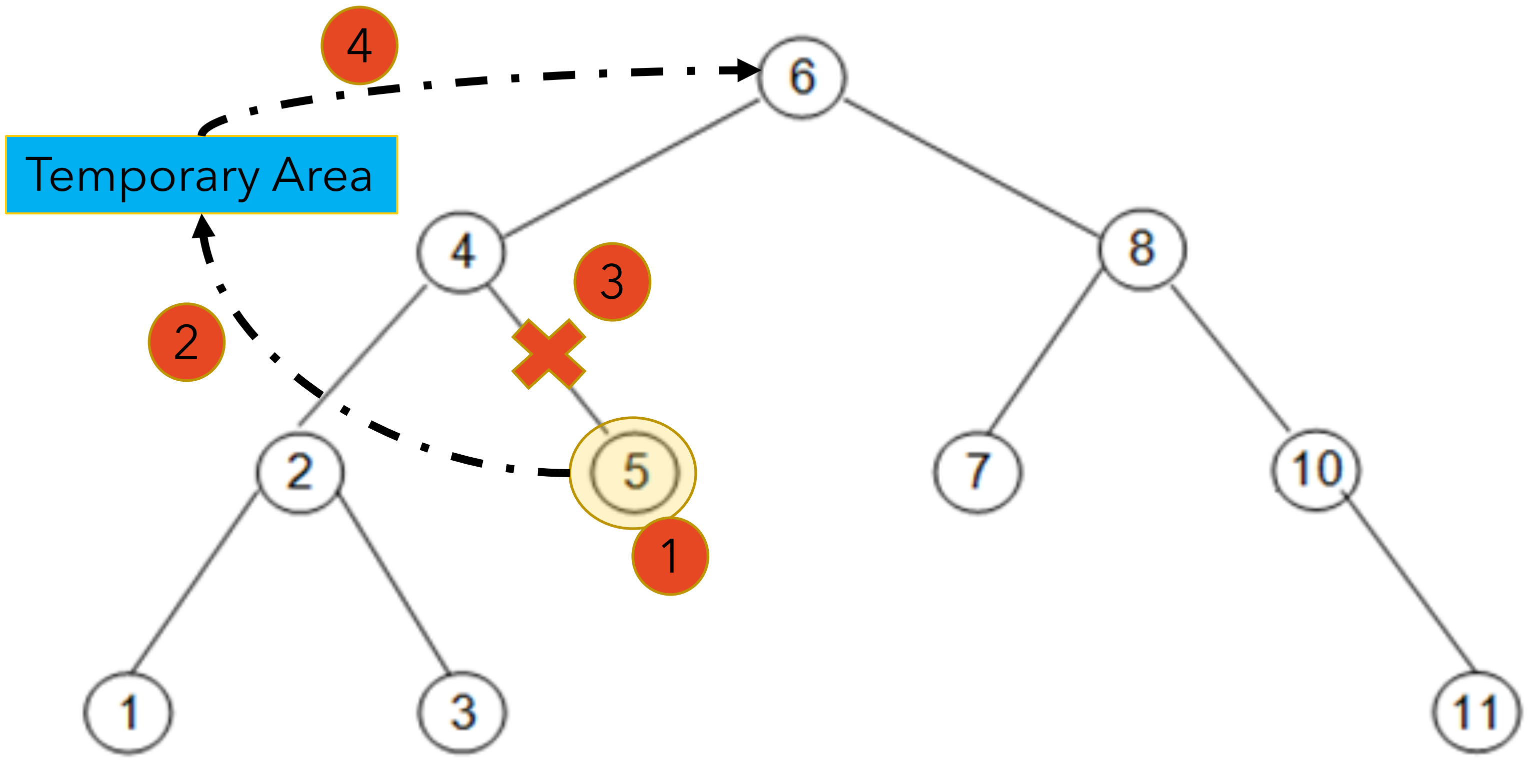
- Case 2: Deleting a node with 1 child
  - Simply delete the node by setting the pointer pointing to it, to point to the node's child (only 1 child)



Binary Search Tree

- Case 3: Deleting a node with 2 children
    1. find the successor (next smaller value) i.e. the rightmost child in the node's left subtree
    2. Store the successor item in a temp variable
    3. delete the successor recursively (either case 1 or case 2)
    4. replace the node's entry with that of the successor (in temp)

Binary Search Tree



Binary Search Tree

- Now let's try to figure out how to code
  - PreOrder traversal
  - PostOrder traversal
  - Search
  - Remove

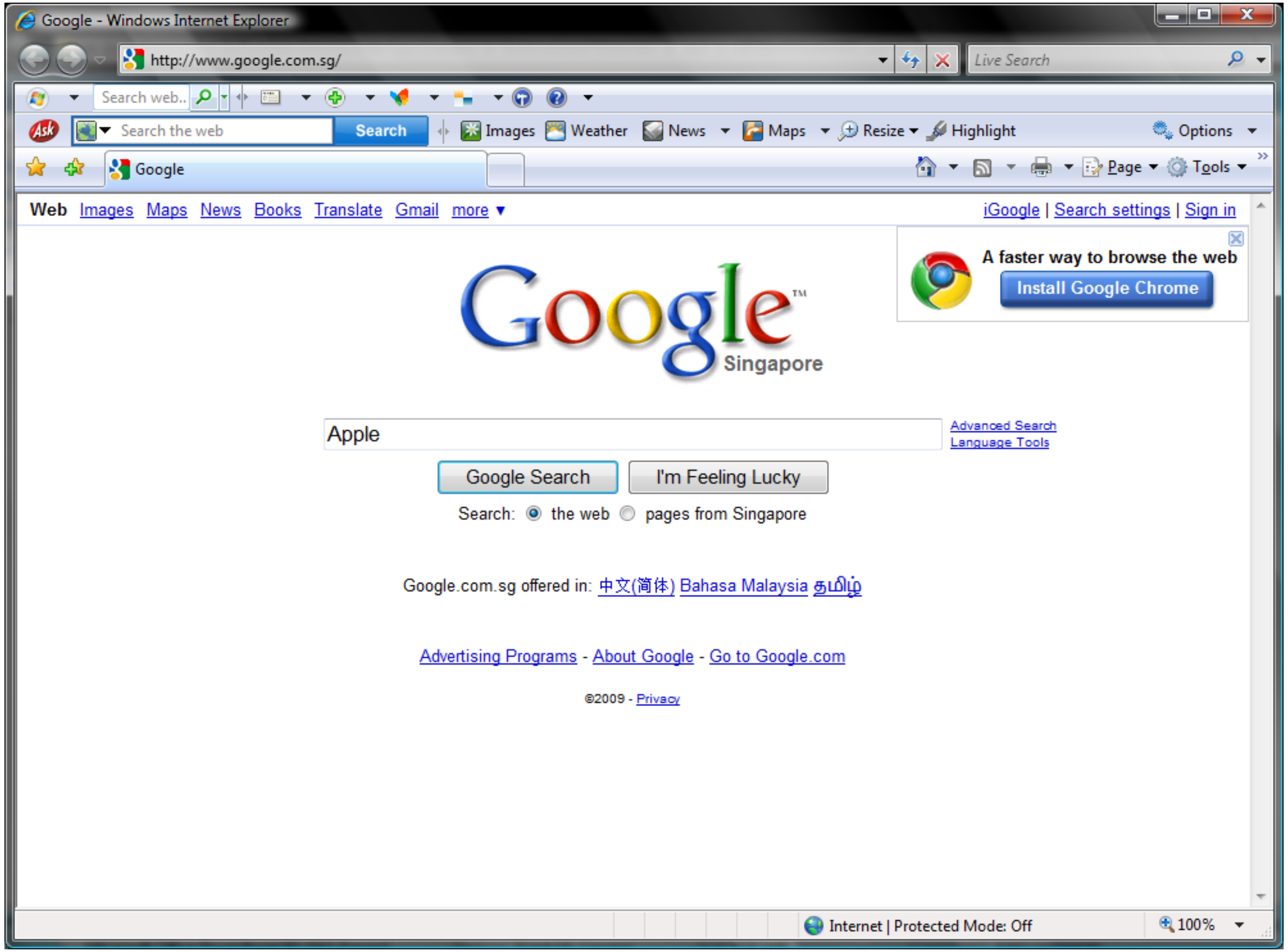
Learning Objectives – Mod 10

At the end of the course, participants should be able to:

- Define what a search algorithm is.
- Examine the different types of search algorithm.
- Demonstrate the different usage of the search algorithm.

Searching

- Searching is one of the most common activities in our daily life.
    - e.g. searching for a book, person, tel no, file, movie, song, . . .



Search

- Types of search
  - Sequential Search – Sorted / Unsorted
  - Binary Search
  - Jump Search
  - Interpolation Search
  - Sublist Search
  - And the list goes on….

Sequential Search - Unsorted

- Given an unsorted array of data, what are the ways to search for "target" data?

0	1	2	3	4	. . .	N-1
48	25	95	76	57	. . .	88

Dave	Eve	Bean	Roy	Ann	. . .	Gary

Search for value 50 or search for John ?

Sequential Search - Unsorted

- Search for 76

Data 1 = 76?

48	25	95	76	57	12	33	88

Data 2 = 76?

48	25	95	76	57	12	33	88

Data 3 = 76?
Data 4 = 76?
Yes, item found, stop

48	25	95	76	57	12	33	88

Sequential Search - Unsorted

- Search for 50

Data 1 = 50?

48	25	95	76	57	12	33	88

Data 2 = 50?
Data 3 = 50?
…
…
Data N = 50?

48	**25**	95	76	57	12	33	88

No and no more data. Stop. Item not found.

Sequential Search - Unsorted

Consider the following method

int search(itemType[] array, int n, ItemType target)

Algorithm:

Set index to start of array.

While(not found and not end of array)

If item at the index of array equal to target

Item found (return index)

Else

Increment the index by 1

End of array reached, item not found, return -1

Sequential Search - Unsorted

- Implementation of said algorithm

```
func search(arrSlice []int, n int, target int) int {
    for i := 0; i < n; i++ {
            if arrSlice[i] == target { // found
            return i
        }
    }
    return -1 // not found
}
```

Sequential Search - Unsorted

- In terms of efficiency for unsorted array
  - Worst number of searches = N
  - Average number of comparisons = N/2

0	1	2	3	4	. . .	N-1
48	25	95	76	57	...	88

Sequential Search - Sorted

- Given a sorted array, what are the ways to search for “target” data?

0	1	2	3	4	5		N-1
12	25	33	48	57	76	...	95

Sequential Search - sorted

- Search for 76

Data 1 = 76?

12	25	33	48	57	76	88	95

Data 2 = 76?

12	**25**	33	48	57	76	88	95

Data 3 = 76?
Data 4 = 76?
Yes, item found, stop

12	25	33	48	57	**76**	88	95

Sequential Search - Sorted

- Search for 50

12	25	33	48	57	76	88	95

Data 1 = 50? No. Data 2 > 50 ? No, continue to next data.
Data 2 = 50? No. Data 2 > 50 ? No, continue to next data.
Data 3 = 50? No. Data 3 > 50 ? No, continue to next data.
Data 4 = 50? No. Data 4 > 50 ? No, continue to next data.
Data 5 = 50? No. Data 5 > 50 ? Yes. Number exceeded. Stop
Item not found.

Sequential Search - Sorted

Consider the following method

int search(itemType[] array, int n, ItemType target)

Algorithm:

Set index to start of array.

While(not found and not done and not end of array)

If item at the index of array equal to target

Item found (return index)

Else if item at index greater than target

Item not found, done. Return -1

Else

Increment the index by 1

End of array reached, item not found, return -1

Sequential Search - Sorted

- In terms of efficiency for unsorted array
  - Worst number of searches = N
  - Average number of comparisons = N/2 (need not search non-relevant items)

0	1	2	3	4	. . .	N-1
12	25	33	48	57	...	95

Binary Search

- Requires a sorted array.
- Uses divide and conquer

0	1	2	3	4	. . .	N-1
12	25	33	48	57	...	95

Binary Search

- Search for 76

mid = (0 + 7) / 2 = 3 (go to somewhere in the middle)

array[3] != 76 (compare the values)

array[3] < 76 (in second half)

0	1	2	3	4	5	6	7
12	25	33	48	57	76	88	95

mid = (4 + 7) / 2 = 5 (go to middle)

array[5] == 76 (compare the values)
found, return 5

4	5	6	7
57	76	88	95

Binary Search

- Search for 80

mid = (0 + 7) / 2 = 3

array[3] != 80

array[3] < 80

0	1	2	3	4	5	6	7
12	25	33	48	57	76	88	95

mid = (4 + 7) / 2 = 5

array[5] != 80

array[5] < 80

4	5	6	7
57	76	88	95

mid = (6 + 7) / 2 = 6

array[6] != 80

array[6] > 80

Not found (first > last) , return -1

6	7
88	95

Binary Search

Consider the following method

int search(itemType[] array, int n, ItemType target)

Algorithm:

Set first to start of array.

Set last to end of array

While (first <= last)

Set mid = (first + last)/ 2

If item at mid is equal to target

Item found, return mid

Else if target is smaller than item at mid

Set last = mid -1

Else

Set first = mid +1

End of search, item not found. Return -1

Binary Search

```
func binarySearch(arr []int, n int, target int) int {
    first := 0
    last := n

    for first <= last {

        mid := (first + last) / 2
        if arr[mid] == target { // found
        return mid
    } else {
        if target < arr[mid] {
            last = mid - 1 // search first half
        } else {
            first = mid + 1 // search second half
        }
    }
    } //end of for
    return -1 // not found
}
```

Binary Search - Sorted

- In terms of efficiency for unsorted array
  - Worst number of searches = $\log_2 N$
  - Average number of comparisons = $(\log_2 N)/2$

0	1	2	3	4	. . .	N-1
12	25	33	48	57	...	95

Sequential vs Binary

N	Sequential Search O(n)	Binary Search $O(\log_2 n)$
1	1	n=2 ->1
4	4	2
8	8	3
16	16	4
32	32	5
64	64	6
128	128	7
256	256	8
512	512	9
1024	1024	10
1000000	1000000	20

Search

Search	Array	Linked-List
Sequential	✓	✓
Binary	✓	×

- Sequential Search can be used to search for data in arrays and linked-lists
  Binary Search can be used to search for data in arrays but not linked-lists

- Using Binary search on arrays instead of Sequential search
  - Improves the time performance from O(n) to O(log n).

- Binary search on linked list
  - Locating the mid in every repetition is non-trival
  - Traversal from the first to the middle node is necessary, taking O(n) time
  - Eventually gives O(n log n), with log n recursive calls/repetitions.

Learning Objectives – Mod 11

At the end of the course, participants should be able to:

- Define what a sort algorithm is.
- Examine the different types of sort algorithm.
- Demonstrate the different usage of the sort algorithm.

Sorting

- ***Sorting***
  - a process that organizes a collection of data
  - Either ascending or descending order.
- ***Sort Key***
  - part of the data item that we would consider during sorting.
    - sorting VIP customers based on amount spent
    - sorting hotel room bookings based on prices
    - sorting popular videos based on number of views

Sorting

- Types of Sorting
  - Selection
  - Insertion
  - Merge
  - Quick
  - Radix

Selection Sort

- *Select the largest item, put in its correct place by swapping*
- Find the *next largest item*, put in its correct place, and so on until array is sorted
- To put an element in its correct place, it *swaps* position with that element in that location.
- The array will have a <u>sorted section</u> that grows from end of array and rest of array remain unsorted.

** can also sort by smallest item*

Selection Sort

Example of a selection sort of an array of five integers.

Shaded elements are selected;
boldface elements are in order.

Initial array:	29	10	14	37	13
After 1st swap:	29	10	14	13	**37**
After 2nd swap:	13	10	14	**29**	**37**
After 3rd swap:	13	10	**14**	**29**	**37**
After 4th swap:	**10**	**13**	**14**	**29**	**37**

Selection Sort

Consider method

void selectionSort(ItemType array[], int n)

Algorithm:

```
let last be index of the last item in the subarray of items yet to be sorted

for (last = n-1; last >= 1; last--)
{
    select largest item in array[0..last]
    swap largest item with array[last]
}
```

Selection Sort

```
func selectionSort(arr []int, n int) {

    for last := n - 1; last >= 1; last-- {

        // select largest item in array[0..last]
        largest := indexOfLargest(arr, last+1)

        // swap largest item array[largest] with array[last]
        swap(&arr[largest], &arr[last])
    }

}
```

Selection Sort

```
func indexOfLargest(arr []int, n int) int {

    largestIndex := 0 // index of  largest item

    for i := 1; i < n; i++ {

        if arr[i] > arr[largestIndex] {
            largestIndex = i
        }
    }

    return largestIndex
}
```

```
func swap(x *int, y *int) {
    temp := *x
    *x = *y
    *y = temp

}
```

Selection Sort

- Worst case: $O(n^2)$
  - *a loop for finding largest within a loop for selection sort, with number of iterations depending on n*

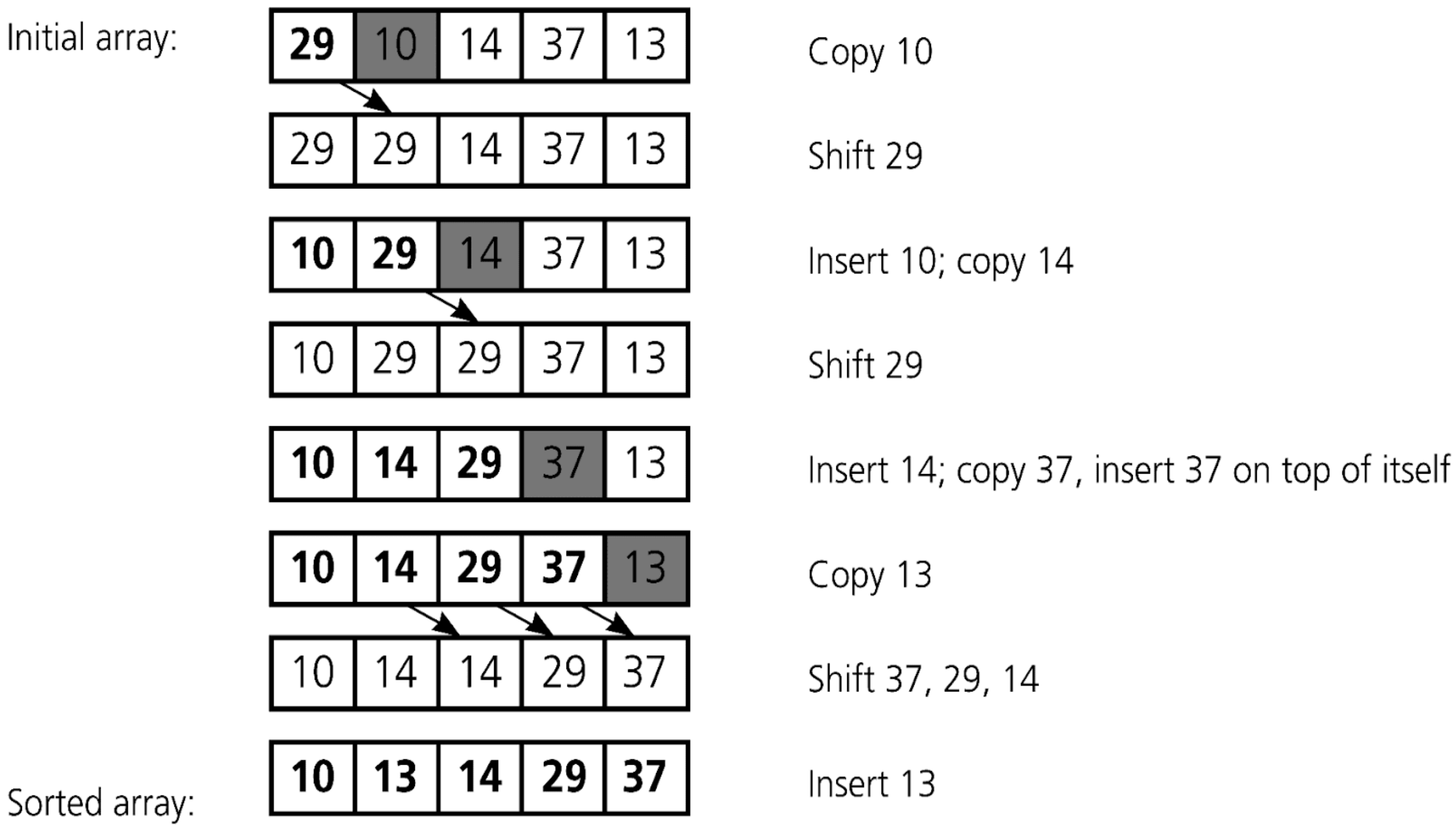
- Average case: $O(n^2)$
- Does not depend on the initial arrangement of the data
- Only appropriate for *small n*

Insertion Sort

- Partition the array into two regions: sorted and unsorted
- Take each item from the unsorted region and insert it into its correct order in the sorted region (need to shift elements)

Insertion Sort

- Example of an insertion sort of an array of five integers.



Source: Copyright © 2007 Pearson Education, Inc. Publishing as Pearson Addison-Wesley. Ver. 5.0.

Insertion Sort

Consider method

void insertionSort(ItemType array[], int n)

Algorithm:

let unsorted be first index of the unsorted region

in general, sorted region is array[0..unsorted-1],
unsorted region is array[unsorted..n-1]
for (int unsorted = 1; unsorted < n; unsorted++)
{
let nextItem be first item in unsorted region
let loc be index of insertion in the sorted region,
find the loc in sorted portion for nextItem;
shift, if necessary, to make room
}

Insertion Sort

```
func insertionSort(arr []int, n int) {
    for i := 1; i < n; i++ { // sorted: 0..i-1  unsorted: i..n-1

        // 1. copy data
        data := arr[i] // 1st elt in unsorted region

        // 2. shift larger data to the right
        last := i
        for (last > 0) && (arr[last-1] > data) {
            arr[last] = arr[last-1]
            last--
        }

        // 3. insert data
        arr[last] = data
    }
}
```

Insertion Sort

- Worst case: $O(n^2)$
  - *(a loop within a loop with no. of iterations depending on n)*

- Average case: $O(n^2)$
- Best case, when items are already sorted: $O(n)$
  - *(array[loc-1] > nextItem condition is always false)*

- Appropriate for small arrays due to its simplicity
- Prohibitively inefficient for large arrays

Insertion Sort

- **Strengths**
  - Good when unordered list is mostly sorted
  - Need minimum time to verify if list is sorted
  - Better with pointer-based implementation (no movement of data)
- **Weaknesses**
  - Every new insertion requires movements/shifting for some inserted items in ordered portion
  - When each slot contains large record => movement is expensive
  - Array-based implementation is less suitable.

Merge Sort

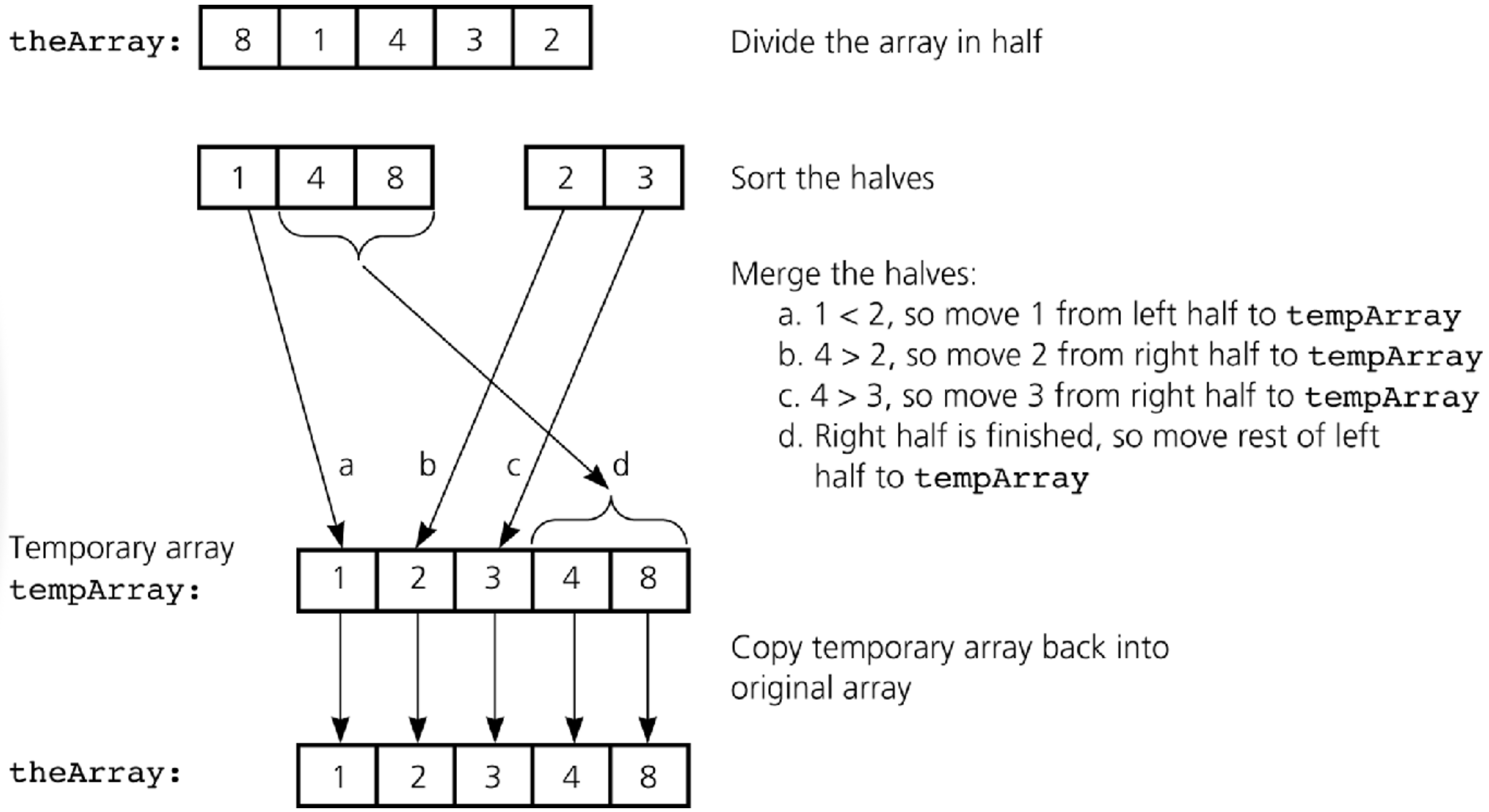
- A recursive sorting algorithm

Strategy

- *Divide* an array into 2 halves
- Sort each half
- *Merge* the sorted halves into one sorted array
- *Divide-and-conquer*

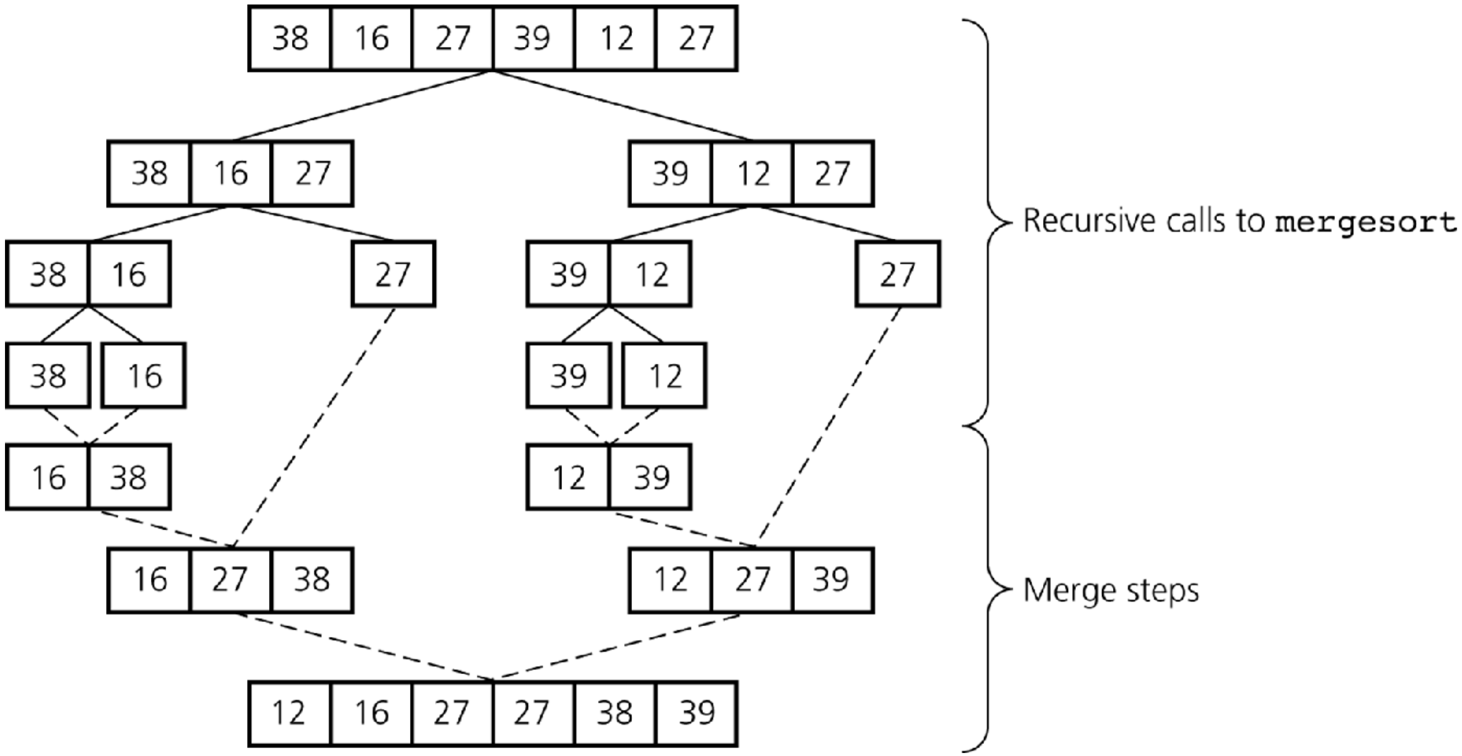
Merge Sort

- Example of a merge sort with an array of intergers



Merge Sort

- Further examples



Merge Sort

Consider method

void mergeSort(ItemType array[], int first, int last)

Algorithm:

```
if (first < last)    // more than 1 items
{
        find index of midpoint
        sort left half array[first..mid]         // recursively
        sort right half array[mid+1..last]    // recursively
        merge the two halves
}
```

Merge Sort

```
func mergeSort(arr []int, first int, last int) {

    if first < last { // more than 1 items

        mid := (first + last) / 2    // index of midpoint
        mergeSort(arr, first, mid)   // sort left half
        mergeSort(arr, mid+1, last)  // sort right half
        merge(arr, first, mid, last) // merge the two halves
    }

}
```

Merge Sort

```
func merge(arr []int, first int, mid int, last int) {

    const maxSize int = 8
    var tempArr [maxSize]int // temporary array
    // initialize the local indexes to indicate the subarrays
    first1 := first   // beginning of first subarray
    last1 := mid      // end of first subarray
    first2 := mid + 1 // beginning of second subarray
    last2 := last     // end of second subarray
```

Merge Sort

```
  // while both subarrays are not empty, copy the
// smaller item into the temporary array
index := first1 // next available location in tempArray
for (first1 <= last1) && (first2 <= last2) {

    if arr[first1] < arr[first2] {
        tempArr[index] = arr[first1]
        first1++
    } else {
        tempArr[index] = arr[first2]
        first2++
    }
    index++
}
```

Merge Sort

```
// finish off the nonempty subarray
// finish off the first subarray, if necessary
for first1 <= last1 {
    tempArr[index] = arr[first1]
    first1++
    index++
}
// finish off the second subarray, if necessary
for first2 <= last2 {
    tempArr[index] = arr[first2]
    first2++
    index++
}
```

Merge Sort

```
    // copy the result back into the original array
    for index = first; index <= last; index++ {
        arr[index] = tempArr[index]
    }
}
```

Merge Sort

- Performance is independent of initial order of the array items.
- Worst case: $O(n \log_2 n)$
- Average case: $O(n \log_2 n)$
  - $\log_2 n$ *levels of recursive calls*
  - n *key comparisons made at each level*

Merge Sort

Strengths

- an extremely fast algorithm, good runtime behaviour
- implements easily when using pointer-based implementation

Weaknesses

- for array implementation, need auxillary storage
- requires data movements again during merging
- costly to implement for array implementation

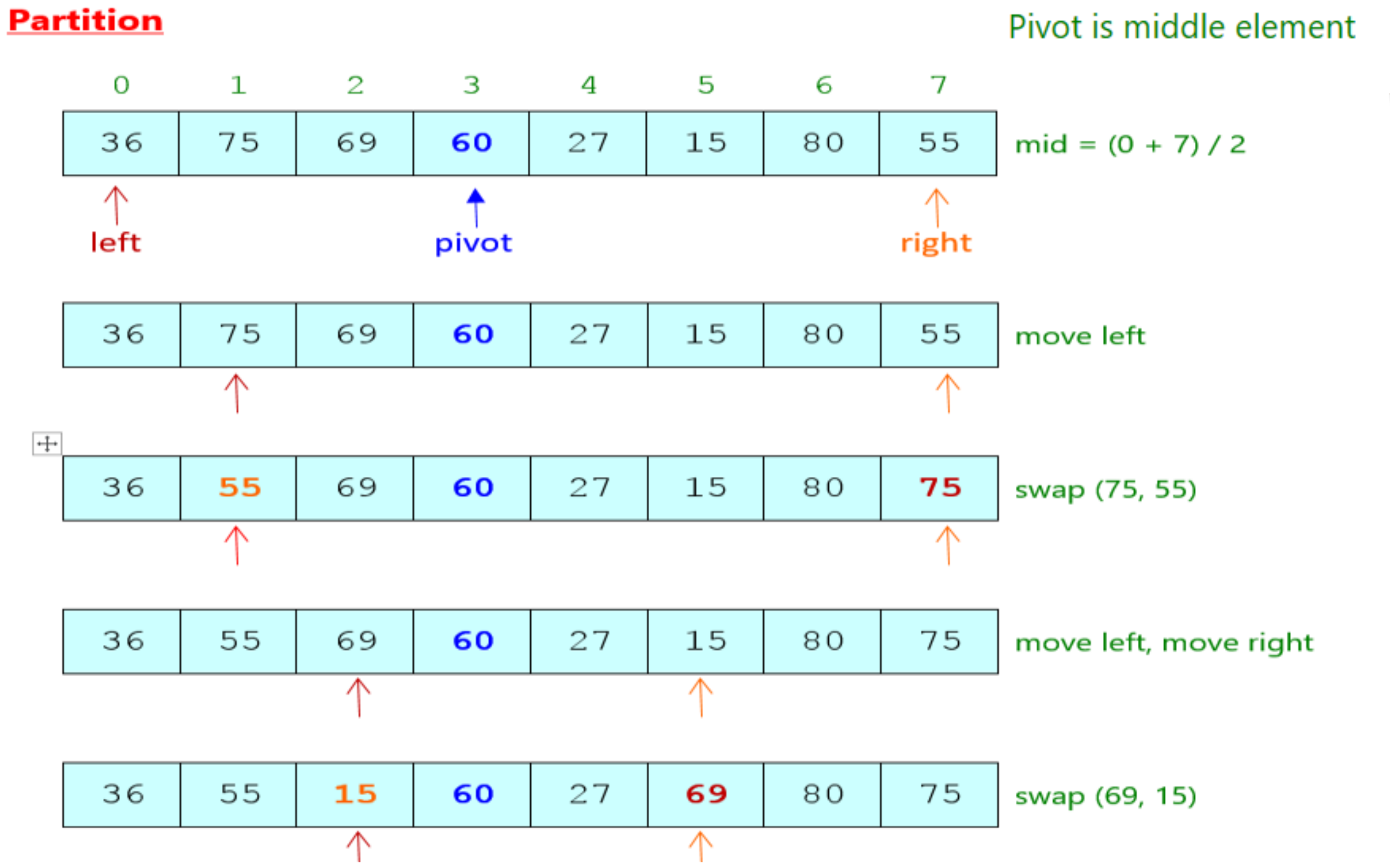
Quick Sort

- Fastest general purpose in-memory sorting algorithm in the average case

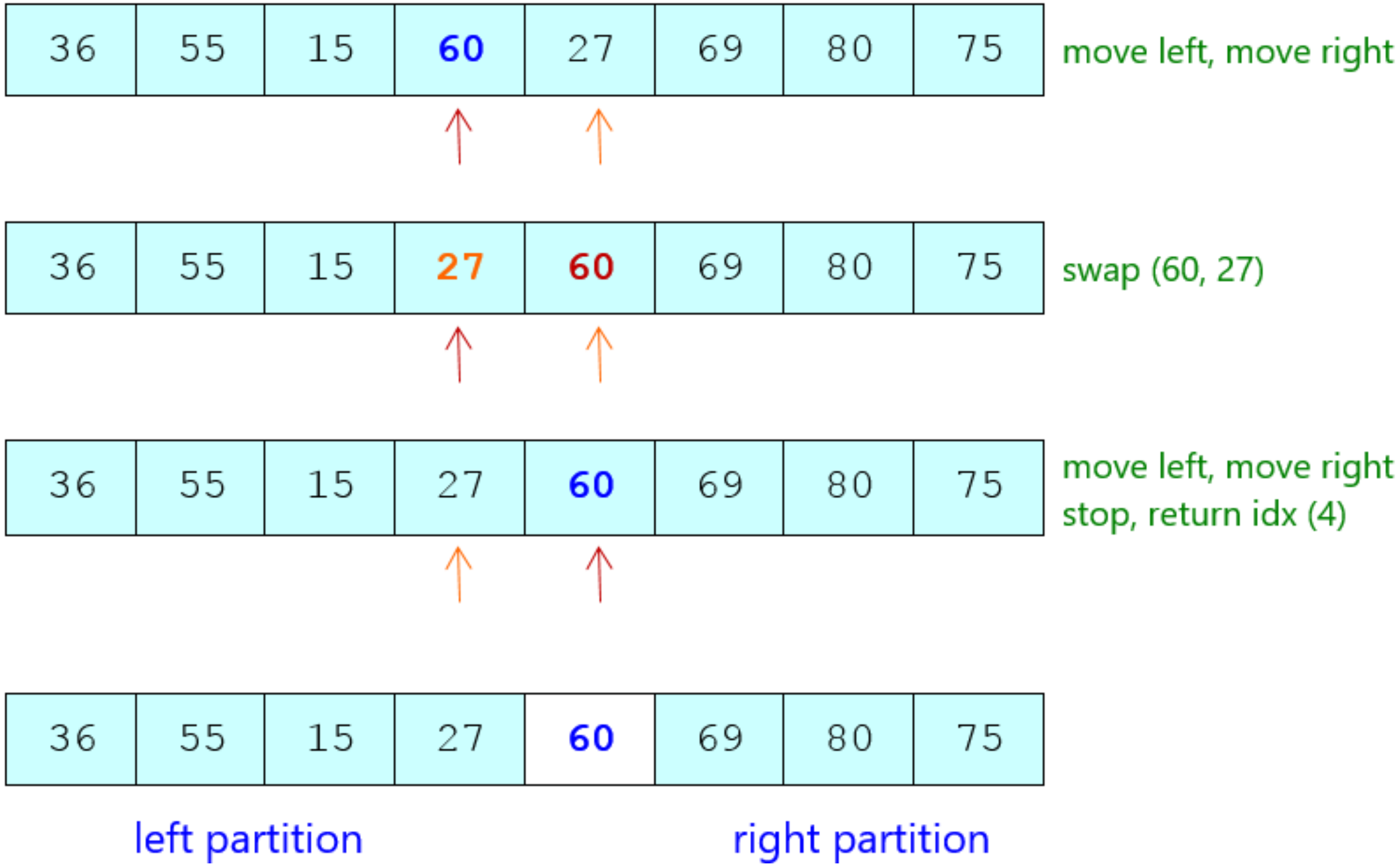
Strategy

- A ***divide-and-conquer*** algorithm
    - ***Partition*** the array into 2 partitions about the pivot
        - Choose an element as ***pivot*** *(can be any element!)*
        - Elements in each part are rearranged such that
            - items < pivot on left of pivot (left partition)
            - items >= pivot on right of pivot (right partition)
- **Recursively** partition the left and right partitions
- (until the partition has less than 2 elements)

Quick Sort


Partition
Pivot is middle element
0 1 2 3 4 5 6 7
36 75 69 60 27 15 80 55
mid = (0 + 7) / 2
left
pivot
right
36 75 69 60 27 15 80 55
move left
36 55 69 60 27 15 80 75
swap (75, 55)
36 55 69 60 27 15 80 75
move left, move right
36 55 15 60 27 69 80 75
swap (69, 15)

Quick Sort


36 55 15 60 27 69 80 75
move left, move right
36 55 15 27 60 69 80 75
swap (60, 27)
36 55 15 27 60 69 80 75
move left, move right
stop, return idx (4)
36 55 15 27 60 69 80 75
left partition
right partition

Quick Sort

- Partition

36	55	15	27	**60**	69	80	75

left partition right partition

- Call quicksort to sort:
  - Left partition recursively
  - Right partition recursively

Quick Sort

Consider method

void quickSort(ItemType array[], int left, int right)

Algorithm:

```
if (first < last)    // more than 1 items
{
        create the partition: S1, pivot, S2
        sort partition S1 //recursively
        sort partition S2 //recursively
}
```

Quick Sort

```
func quickSort(array []int, left int, right int) {
    if left < right { // stop when there are less than 2 elements

        pivotIdx := partition(array, left, right)
        quickSort(array, left, pivotIdx-1)  // sort left partition
        quickSort(array, pivotIdx+1, right) // sort right partition
    }

}
```

Quick Sort

```
func partition(arr []int, left int, right int) int {

    pivot := arr[(left+right)/2] // can be any element!

    for left < right { // stop when there are less than 2 elements

        for arr[left] < pivot { // move left index
            left++
        }
        for arr[right] > pivot { // move right index
            right--
        }
        if left < right { // swap the elements
            temp := arr[left]
            arr[left] = arr[right]
            arr[right] = temp
        }
    }
    return left // return the pivot index
}
```

Quick Sort

- Average case: $O(n \log_2 n)$
  - $\log_2 n$ *levels of recursive calls*
  - n *key comparisons made at each level)*
- Best case: $O(n \log_2 n)$
  - *When pivot always splits array into equal halves*
- Worst case: $O(n^2)$
  - *When at each recursive call, the smallest or biggest item is chosen as the pivot*

Quick Sort

Strengths

- *Fast on average*
- No merging required
- *Best case* if pivot always splits data into *equal halves*

Weaknesses

- Pivot need to be chosen carefully
- Performs badly when
    - *array is already sorted* and *pivot is largest or smallest* element
    - *n is small*
- *due to overhead in recursive calls*

Radix Sort

- Treats each data element as a character string
- Repeatedly organizes the data into groups according to the i^{th} character/digit in each element

Radix Sort

0123	2154	0222	0004	0283	1560	1061	2150	Original data
(1560,	2150)	(1061)	(0222)	(0123,	0283)	(2154,	0004)	Group by 4th digit
1560	2150	1061	0222	0123	0283	2154	0004	Combined
(0004)	(0222, 0123)		(2150, 2154)		(1560, 1061)		(0283)	Group by 3rd digit
0004	0222	0123	2150	2154	1560	1061	0283	Combined
(0004, 1061)		(0123, 2150, 2154)			(0222, 0283)		(1560)	Group by 2nd digit
0004	1061	0123	2150	2154	0222	0283	1560	Combined
(0004,	0123,	0222,	0283)	(1061,	1560)	(2150,	2154)	Group by 1st digit
0004	0123	0222	0283	1061	1560	2150	2154	Combined

Radix Sort

Consider method

```
void radixSort(ItemType array[], int n, int digit)
```

Algorithm:

```
for(j = digit down to 1)
{
    initialize 10 groups to empty
    initialize a counter for each group to 0

    for (i = 0 through n-1)
    {
        k = jth digit of array[i]
        place array[i] at the end of group k
        increase kth counter by 1
    }
    replace the items in theArray with all the items in group 0,
    followed by group 1 and so on.
}
```

Radix Sort

- Distributing the elements according to value of digit : `O(n)`
  - *n is the number of iterations*
- Combining the elements at the last step : `O(n)`
- Involves distributing and combining for no of times equal to no of digits
  - *which is a constant, not dependent on n*
- *Overall complexity is* `O(n)`
  - key is treated as string and must have the same length
  - restrictive, not universal

Comparing Sorting Algorithm

- Approximate complexity of the different types of sorting algorithm.

Time Complexity	Worse Case	Average Case
Selection Sort	$O(n^2)$	$O(n^2)$
Insertion Sort	$O(n^2)$	$O(n^2)$
Merge Sort	$O(n \log_2 n)$	$O(n \log_2 n)$
Quick Sort	$O(n^2)$	$O(n \log_2 n)$
Radix Sort	$O(n)$	$O(n)$
Tree Sort	$O(n^2)$	$O(n \log_2 n)$