UNIT-1

Abstract Data Type (ADT)

Specification and Design Techniques
Elementary ADTs: Lists, Trees, Stacks, Queues, and Dynamic Sets.

Abstract Data Types (ADTs)

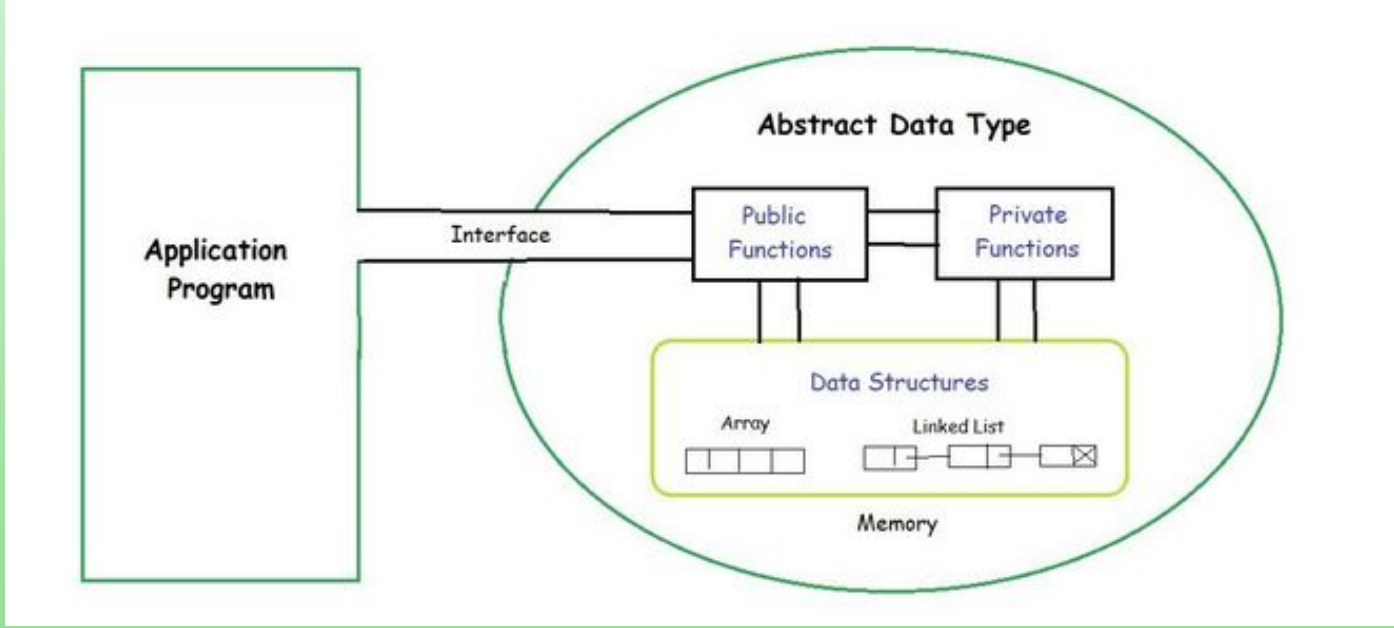
- An abstract data type (ADT) is an abstraction of a data structure

- An ADT specifies:
  - Data stored
  - Operations on the data
  - Error conditions associated with operations

Abstract Data Type (ADT)

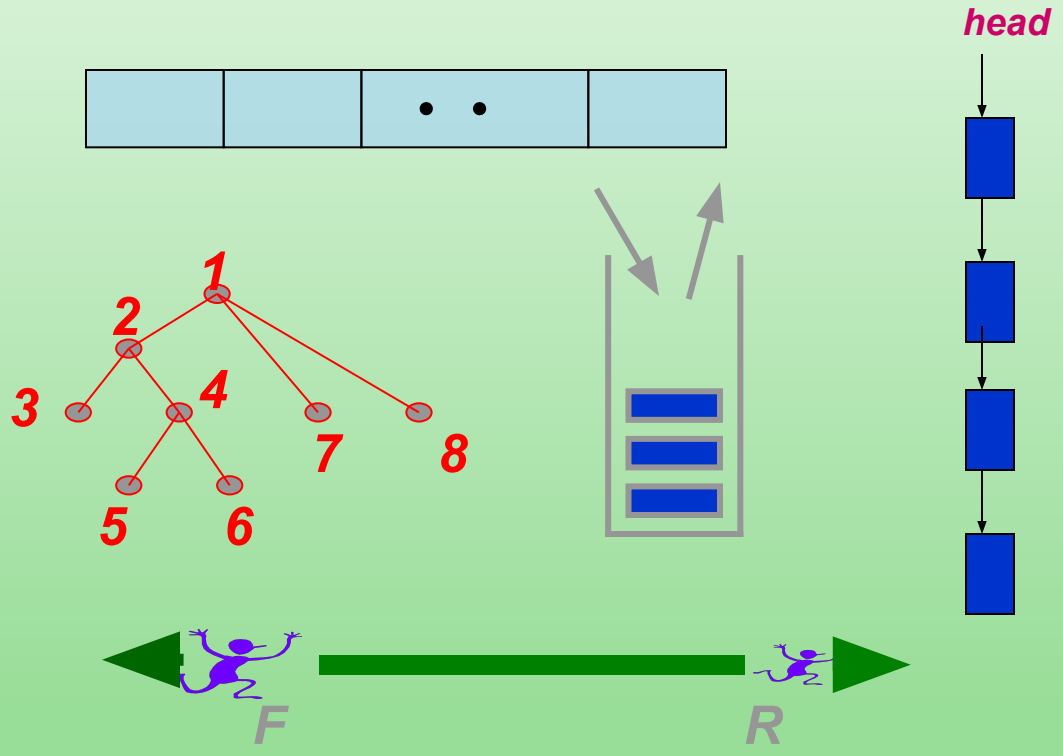
- Definition: a collection of *data* together with a set of *operations* on that data
  - specifications indicate *what* ADT operations do, but not *how* to implement them
  - data structures are part of an ADT's implementation
- Programmer can use an ADT without knowing its implementation.

Illustration- ADTs



Elementary Data “Structures”

- **Arrays**
- **Lists**
- **Stacks**
- **Queues**
- **Trees**



In some languages these are basic data types – in others they need to be implemented

Abstract Data Types Overview

Abstract Data Type	Other Common Names	Commonly Implemented with
List	Sequence	Array, Linked List
Queue		Array, Linked List
Double-ended Queue	Dequeue, Deque	Array, Doubly-linked List
Stack		Array, Linked List
Associative Array	Dictionary, Hash Map, Hash, Map**	Hash Table
Set		Red-black Tree, Hash Table
Priority Queue	Heap	Heap

ADT: Array

C++

```cpp
/* Static Allocation (will be allocated in the stack)*/
int arr[5];
arr[3] = 5;

/* Dynamic Allocation  (will be allocated in the heap)*/
int *arr = new int[5];
arr[3] = 5;

/* ... */
/* Remember to de-allocate memory allocated in the heap when you're done.*/
delete [] arr;
```

Java

```java
int[] newList = new int[5];
newList[2] = 4;
```

ADT: Array

Function Name*	Provided Functionality
`set(i, v)`	Sets the value of index *i* to *v*
`get(i)`	Returns the value of index *i*

ADT: Array

1. How would you print out every element in an array?

2. How would you determine if a certain value is in an array? (Note: This is sometimes called search.)

3. How would you add two arrays together? (Hint: You have to create a new array.)

Ans 1.
We know the size of our static array. Let's say it's 10. Then, we simply run our get(i) function for every index from 1 to 10.

Ans 2.
We know the size of our static array. Let's say it's 10. Then, we simply run our get(i) function for every index from 1 to 10.

Ans 3.
Let's say we have two arrays, A and B. A has size 4 and B has size 6.
If we want to do the operation A + B, we need to create an array C with size 10.
Then, we can search through every element in A and add them to the corresponding index in C.
Then, we'll look through every element in B and add them to the corresponding index plus 4 in C.

```
1  >>> A = [0,2,4,6]
2  >>> B = [1,3,5,7,9,11]
3  >>> C = A + B
4  >>> C
5  [0,2,4,6,1,3,5,7,9,11]
```

The Java array type is an example of an implementation of the array abstract data type that uses an array data structure. Other implementations exist in other languages that we'll look at later.

Creating an array:

Arrays require a size and type when they are created in Java. Below is an example of how you would declare an array of integers with size 10.

EXAMPLE

Creating an array

Java

```
>>> int[] myArray = new int[10];
>>> System.out.print(Arrays.toString(myArray));
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Notice how the array defaults to using zeros. This is because we created an array of integers. If we created an array of strings, the default value would be `null`.

Java

```
>>> String[] myArray = new String[10];
>>> System.out.print(Arrays.toString(myArray));
[null, null, null, null, null, null, null, null, null, null]
```

Setting values:

To set a value at a certain index into the Java array, you use the `[]` syntax. The index goes inside the brackets, and the value goes on the other side of the equation. In Java, indexes **start at zero!** Remember to do the appropriate research for your programming language to know how it handles indexes.

Java

```
>>> int[] myArray = new int[5];
>>> myArray[1] = 100;
>>> myArray[3] = 200;
>>> System.out.print(Arrays.toString(myArray));
[0, 100, 0, 200, 0]
```

Getting a value:

Getting a value at a certain index uses the same notation as setting a value.

EXAMPLE

Getting an index's value

Java

```
>>> int[] myArray = new int[5];
>>> myArray[1] = 100;
>>> System.out.print(myArray[1]);
100
```

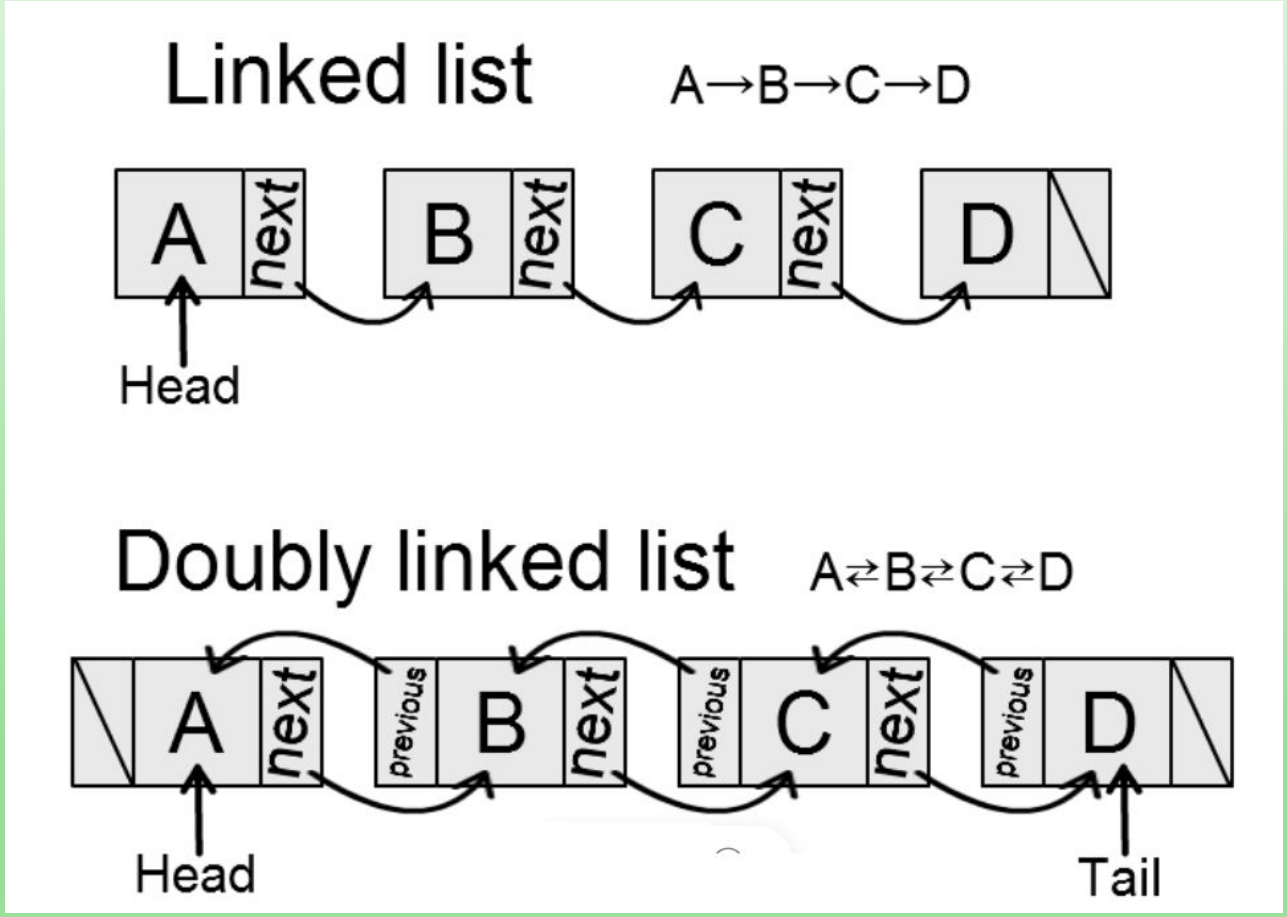
Linked Lists

Lists

- List: a finite sequence of data items
  a1, a2, a3, …, an

- Typical operations:
  - Creation
  - Insert / remove an element
  - Test for emptiness
  - Find an item/element
  - Current element / next / previous
  - Find k-th element
  - Print the entire list

Linked Lists



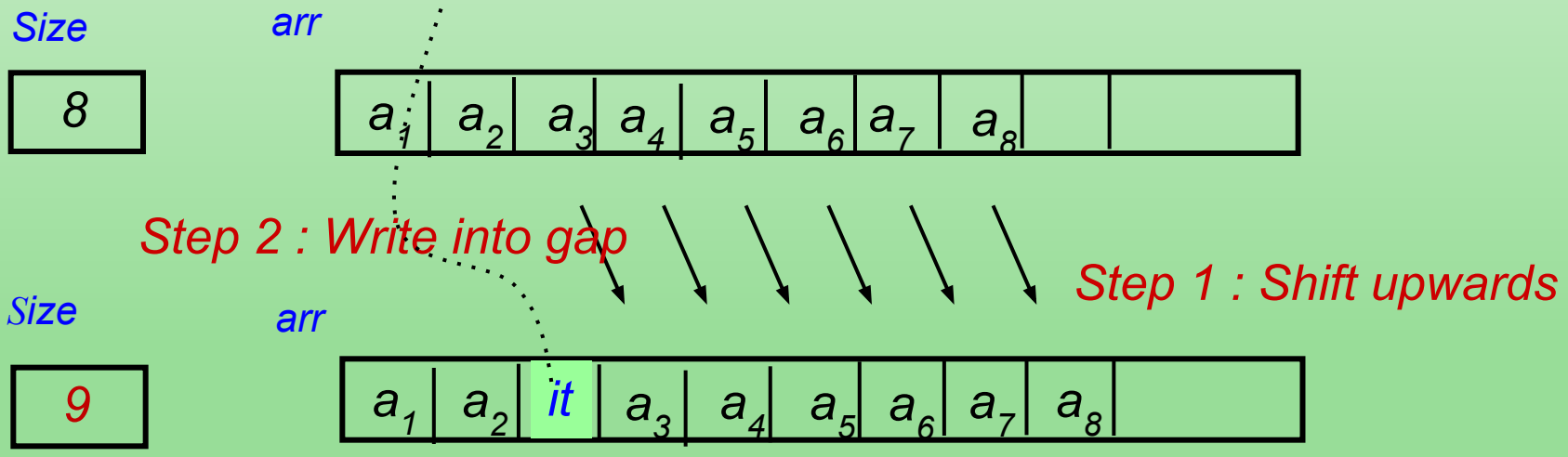
Array-Based List Implementation

- One simple implementation is to use arrays
  - A sequence of n-elements
- Maximum size is anticipated apriori.
- Internal variables:
  - Maximum size *maxSize* (m)
  - Current size *curSize* (n)
  - Current index *cur*
  - Array of elements listArray

Inserting Into an Array

- While retrieval is very fast, insertion and deletion are very slow
  - Insert has to shift upwards to create gap

Example : insert(2, it, arr)

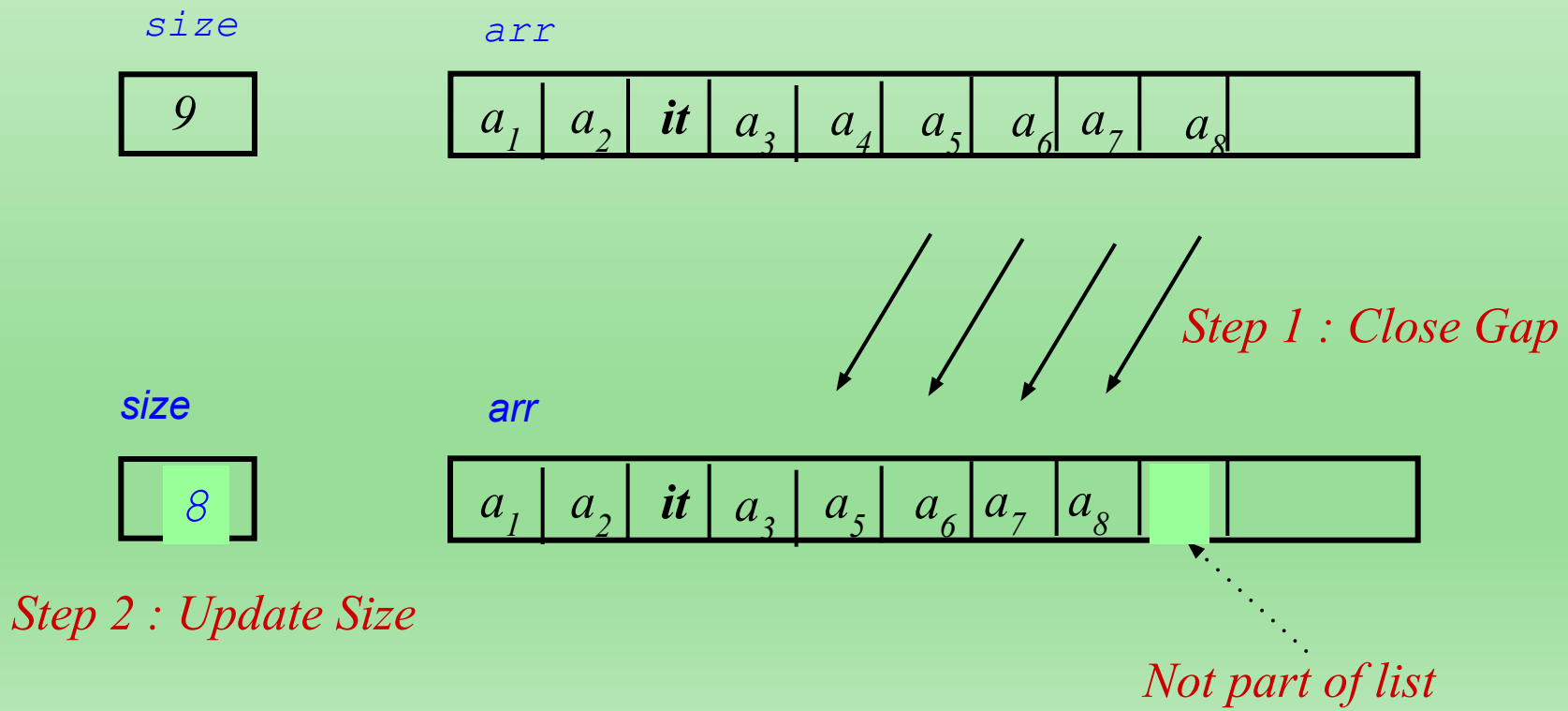


Step 3 : Update Size

Deleting from an Array

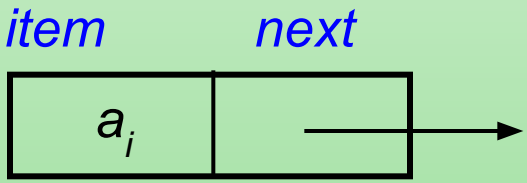
- Delete has to shift downwards to close gap of deleted item

Example: `deleteItem(4, arr)`

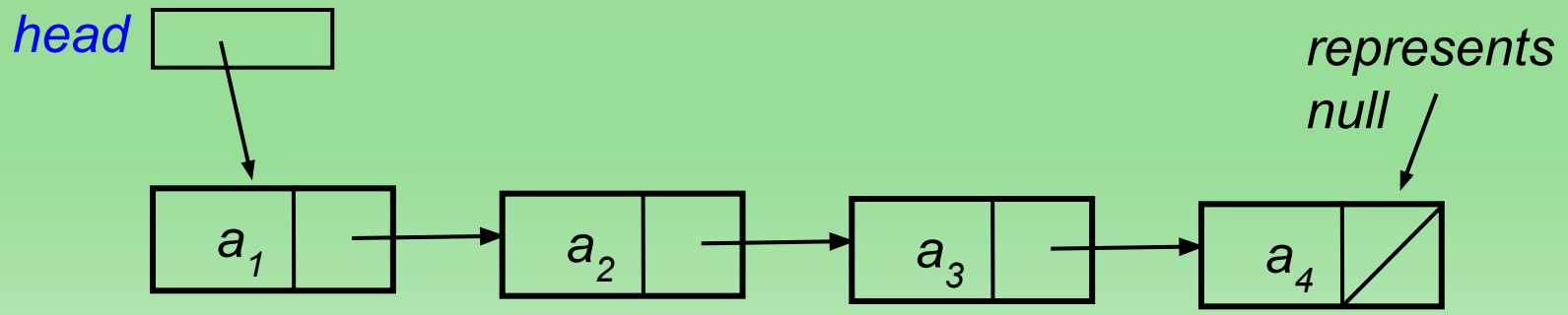


Linked List Approach

- Main problem of array is the slow deletion/insertion since it has to shift items in its *contiguous* memory
- **Solution**: linked list where items need *not be contiguous* with nodes of the form



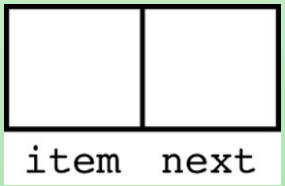
- Sequence (list) of four items < a_1, a_2, a_3, a_4 > can be represented by:



Pointer-Based Linked Lists

- A node in a linked list is usually a struct

```
struct Node
{ int item
  Node *next;
}; //end struct
```



A node

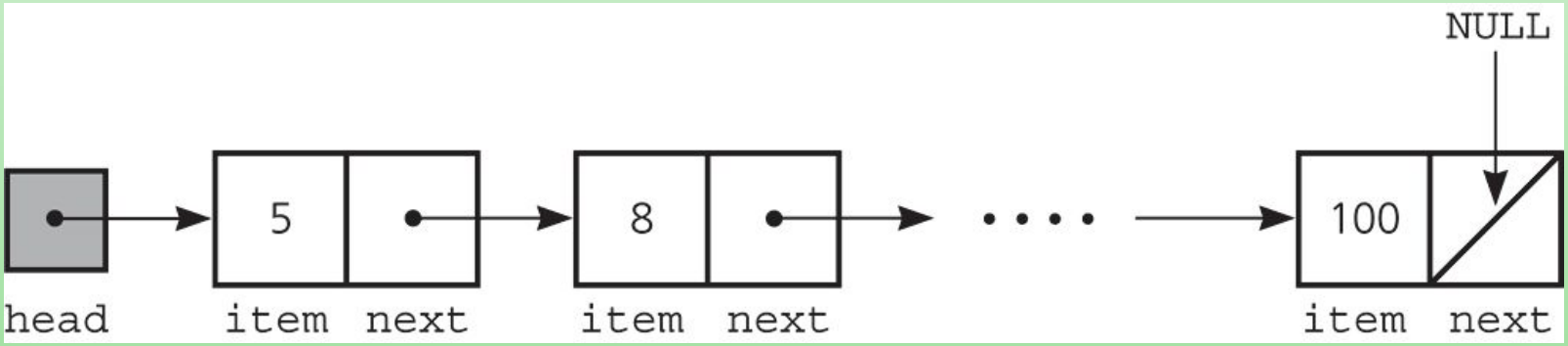
- A node is dynamically allocated

```
Node *p;
p = malloc(sizeof(Node));
```

Pointer-Based Linked Lists

- The head pointer points to the first node in a linked list
- If head is *NULL*, the linked list is empty
  - head=NULL
- head=malloc(sizeof(Node))

A Sample Linked List



An ADT Interface for List

ADT — Interface

The Linked List interface can be implemented in different ways, is important to have operations to `insert` a new node and to `remove` a Node:

```
# Main operations
prepend(value)          -> Add a node in the beginning
append(value)           -> Add a node in the end
pop()                   -> Remove a node from the end
popFirst()              -> Remove a node from the beginning

head()                  -> Return the first node
tail()                  -> Return the last node
remove(Node)            -> Remove Node from the list
```

Traversal – Lists

Traversal : You must be thinking, "how could I print all the elements of a Linked List?"

A Linked List can be `traversed` , is possible to navigate in the list using the nodes next element.

```
list = LinkedList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

currentNode = LinkedList.head()    # Get the first Node

while the currentNode.next
    print currentNode.value

    currentNode = currentNode.next     # Assign the next element
```

Search - Lists

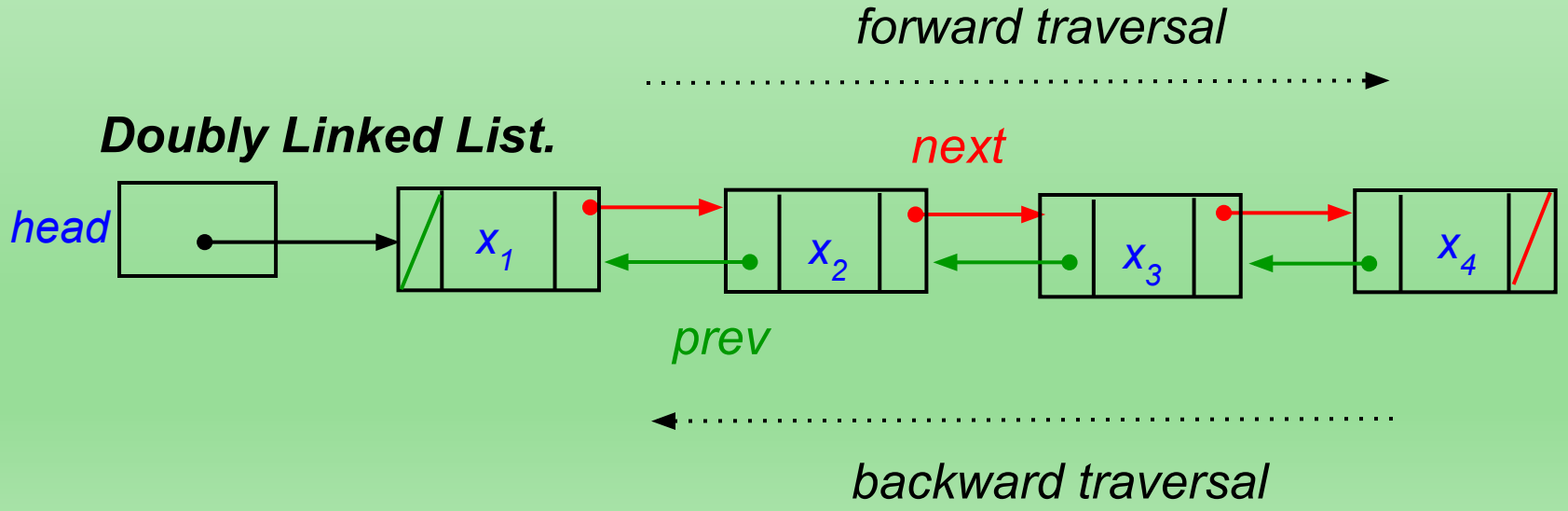
Search: You must be thinking, "how do I find an element inside the Linked List?"

Is possible to `traverse` the array and compare if the element we are looking for exists in the array:

```
list = LinkedList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
targetValue = 7

currentNode = LinkedList.head()    # Get the first Node

while the currentNode.next
    if currentNode.value is equal to targetValue
        print Node found

    currentNode = currentNode.next     # Assign the next element
```

Doubly Liked Lists

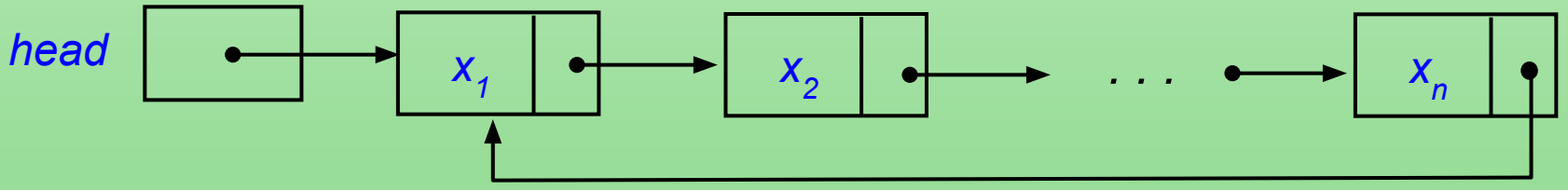
- Frequently, we need to traverse a sequence in BOTH directions efficiently
- *Solution* : Use doubly-linked list where each node has two pointers



Circular Linked Lists

- May need to cycle through a list repeatedly, e.g. round robin system for a shared resource
- *Solution* : Have the last node point to the first node

Circular Linked List.



Time Complexity Lists

- Insertion - O(1)
- Search - O(n)
- Deletion - O(1)
- Indexing - O(n)
- Space - O(n)

New Node creation and initialization

```java
public class Node{
    public int value;
    public Node next;
    public Node(int v , Node n){
        value = v;
        next = n;
    }
}
```

```java
Node one = new Node(6, null)
```

```java
Node two = new Node(4, one);
```

```java
Node three = new Node(5, two)
```

Variable one points to a node that contains the value 6.
The next field of the cell is null, which could indicate the end of the list.

This can be represented as a list with two elements in it by writing

4→6

5→4→6

Linked list construction by reading from an array

EXAMPLE

It is not necessary for us to create a local variable whenever we need to add a new item to a linked list. A linked list can also be constructed by reading from an array.

Java

```java
public static Node createArray(int[] array){
    Node list = null;
    for (int i : array){
        list = new Node(i, list)
    }
    return list;
}
```

Iteration and Recursion in Linked Lists

Iteration and Recursion on Linked Lists

A linked list can be printed either by using recursion or using a loop (iteration). Here is an iterative way of printing a linked list:

Java

```java
public static void printIteratively(Node cell) {
    while (cell != null) {
        System.out.println(cell.value);
        cell = cell.next;
    }
}
```

We can write the same method recursively:

Java

```java
public static void printRecursively(Node cell) {
    if (cell != null) {
        System.out.println(cell.value);
        printRecursively(cell.next);
    }
}
```

Example

EXAMPLE

Write a method that determines the length of a given linked list recursively.

Using our discussed method of iteration, the method can be written as follows:

Java

```java
public static void length(Node cell) {
    int count = 0;
    while (cell != null){
        count += 1;
        cell = cell.next;
    }
}
```

EXAMPLE

Now write a method that determines the length of a linked list recursively.

We can solve this by considering the base case (when P is null) and the recursive case

$$\text{length}(n) = \text{length}(n-1) + 1.$$

Java

```java
public static int length(Node cell) {
    if (cell != null)
        return 1 + length(cell.next);
    return 0;
}
```

Node Delete from Doubly Linked Lists

Java

```java
public class DLL {

    private class Node {
        int value;
        Node next, prev;

        Node (E v, Node n, Node p) {
            value=v; next=n; prev=p;
        }

        Node () {      // used only to create header Node in a list
            // that's initially empty
            next=prev=this;
        }
    }

    private Node header = new Node();

    public boolean isEmpty() {
        return header.next == header;
    }

    private E delete(Node p) {
        p.next.prev = p.prev;
        p.prev.next = p.next;
        return p.value;
    }


}
```

Example

EXAMPLE

Implement a method to add a node to the end of a linked list:

Java

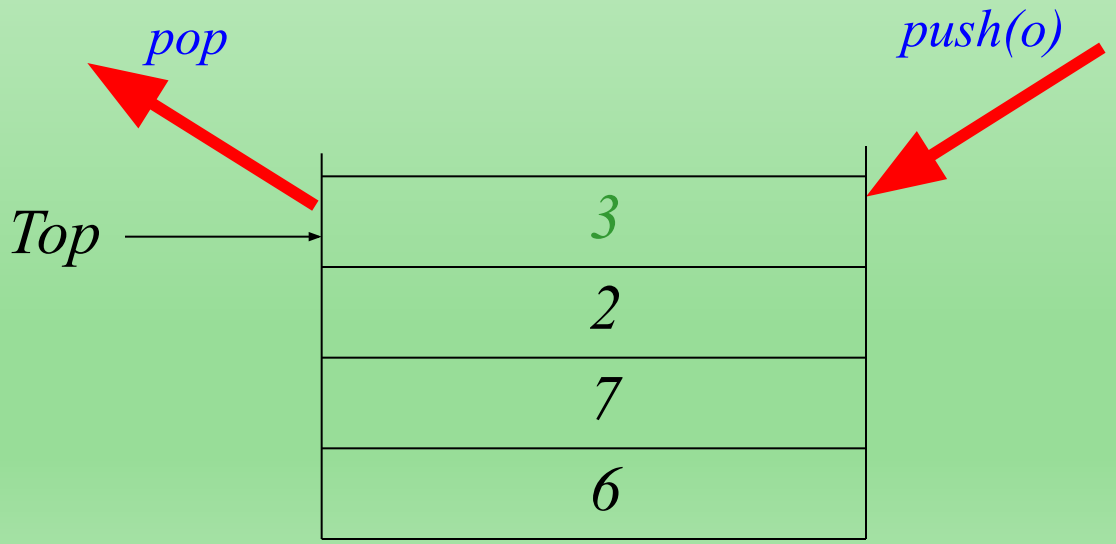
```java
public void addLast(int s){
    Cell p = new Cell(s , header , header.prev);
    header.prev.next = p;
    header.prev = p;
}
```

Stacks

What is a Stack?

- A *stack* is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the *top*.

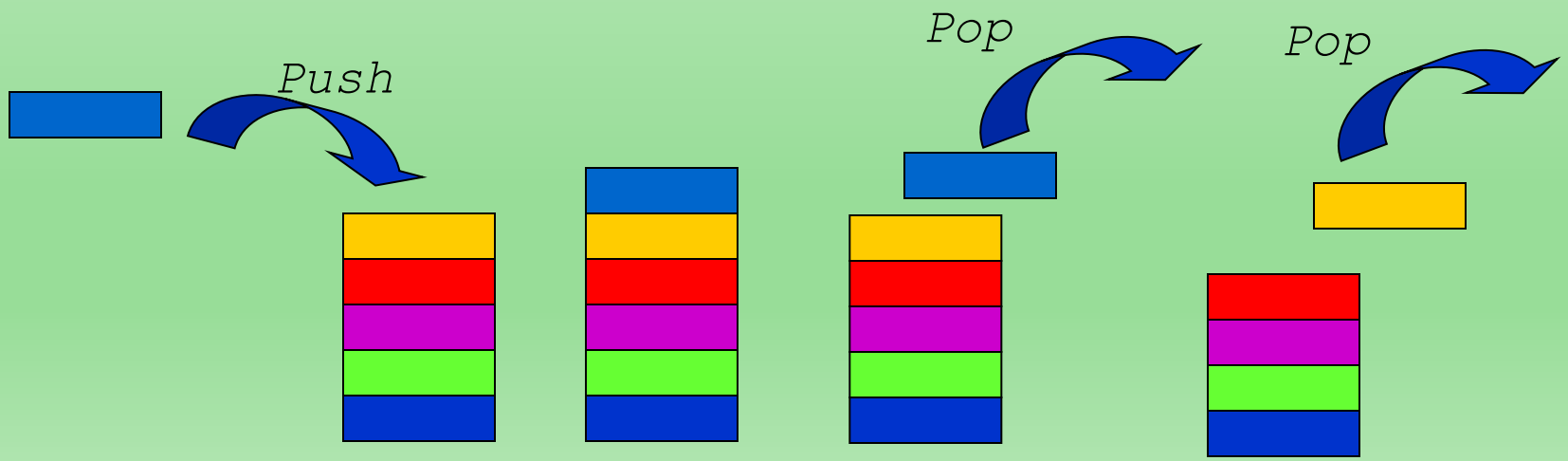
- The operations: push (insert) and pop (delete)



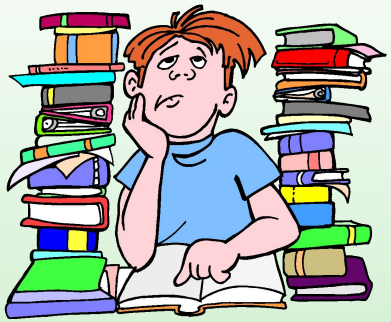
Stack

A list for which Insert and Delete are allowed only at one end of the list (the *top*)

- LIFO – Last in, First out

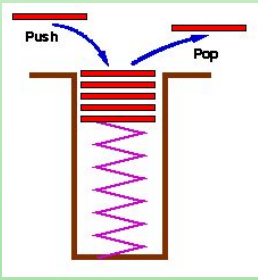


Stack ADT



There are two basic operations related to the stack.

Function Name*	Provided Functionality
`push(i)`	Insert element *i* at the top of the stack.
`pop()`	Remove the element at the top of the stack.



Stacks are often implemented with the following functions.

Function Name*	Provided Functionality
`size()`	returns the current size of the stack.
`peek()`	returns the element at the top of the stack *without changing the stack*

Stack ADT Interface

The main functions in the Stack ADT are (S is the stack)

```
boolean isEmpty();          // return true if empty
boolean isFull(S);          // return true if full
void push(S, item);         // insert item into stack
void pop(S);                // remove most recent item
void clear(S);              // remove all items from stack
Item top(S);                // retrieve most recent item
Item topAndPop(S);          // return & remove most recent item
```

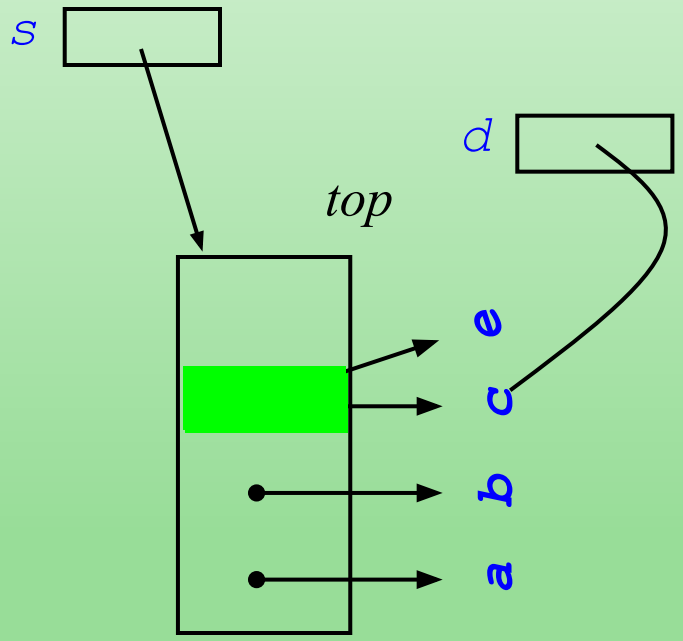
Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception
- Exceptions are said to be “thrown” by an operation that cannot be executed
- In the Stack ADT, operations pop and top cannot be performed if the stack is empty
- Attempting the execution of pop or top on an empty stack throws an EmptyStackException

Sample Operation

Stack S = malloc(sizeof(stack));

- push(S, “a”);
- push(S, “b”);
- push(S, “c”);
- d=top(S);
- pop(S);
- push(S, “e”);
- pop(S);



Array-based Stack

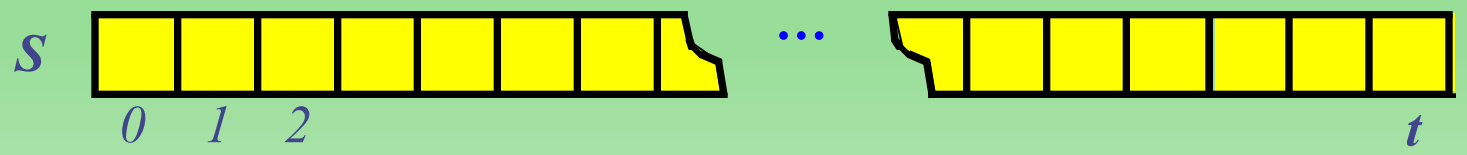
- A simple way of implementing the Stack ADT uses an array
- We add elements from left to right
- A variable keeps track of the index of the top element

```
Algorithm size()
    return t + 1

Algorithm pop()
    if empty() then
        throw EmptyStackException
     else
        t = t - 1
        return S[t + 1]
```

Array-based Stack (cont.)

- The array storing the stack elements may become full
- A push operation will then throw a FullStackException
  - Limitation of the array-based implementation
  - Not intrinsic to the Stack ADT

```
Algorithm push(o)
    if t = S.length - 1 then
        throw FullStackException
    else
        t = t + 1
        S[t] = o
```



Performance and Limitations
- array-based implementation of stack ADT

- Performance
  - Let ***n*** be the number of elements in the stack
  - The space used is ***O***(***n***)
  - Each operation runs in time ***O***(1)
- Limitations
  - The maximum size of the stack must be defined *a priori*, and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

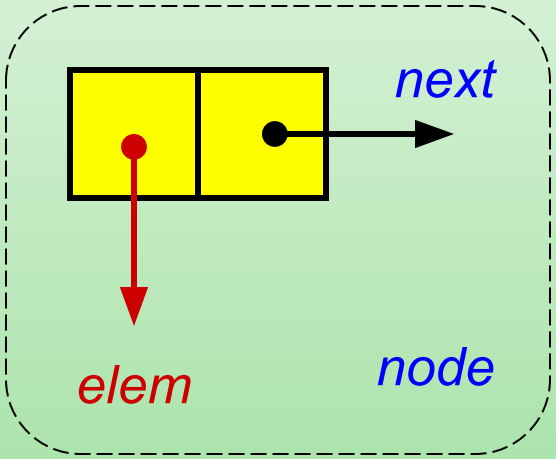
Growable Array-based Stack

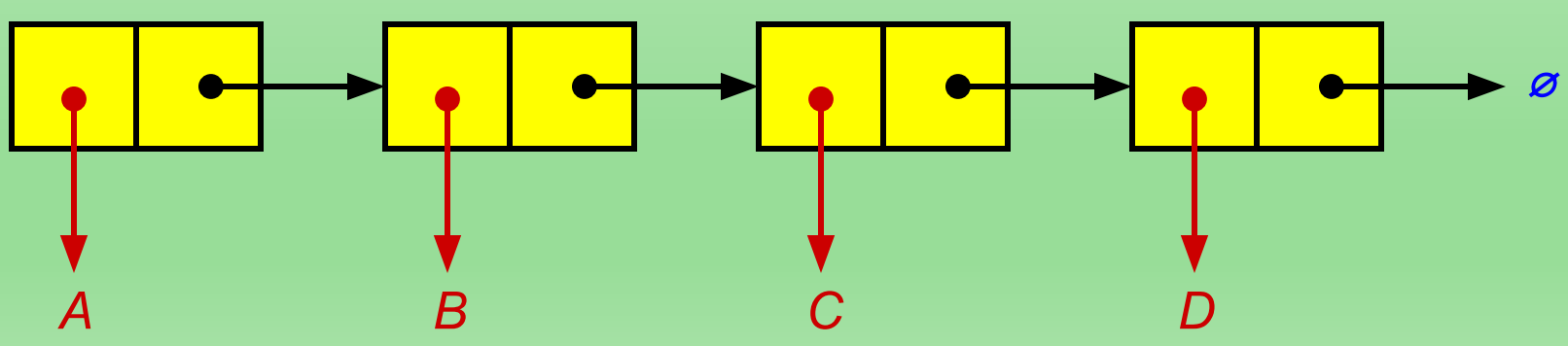
- In a push operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- How large should the new array be?
  - incremental strategy: increase the size by a constant ***c***
  - doubling strategy: double the size

```
Algorithm push(o)
    if t = S.length - 1
then
        A = new array of
                    size …
        for i = 0 to t do
            A[i] = S[i]
            S = A
    t = t + 1
    S[t] = o
```

Singly Linked List

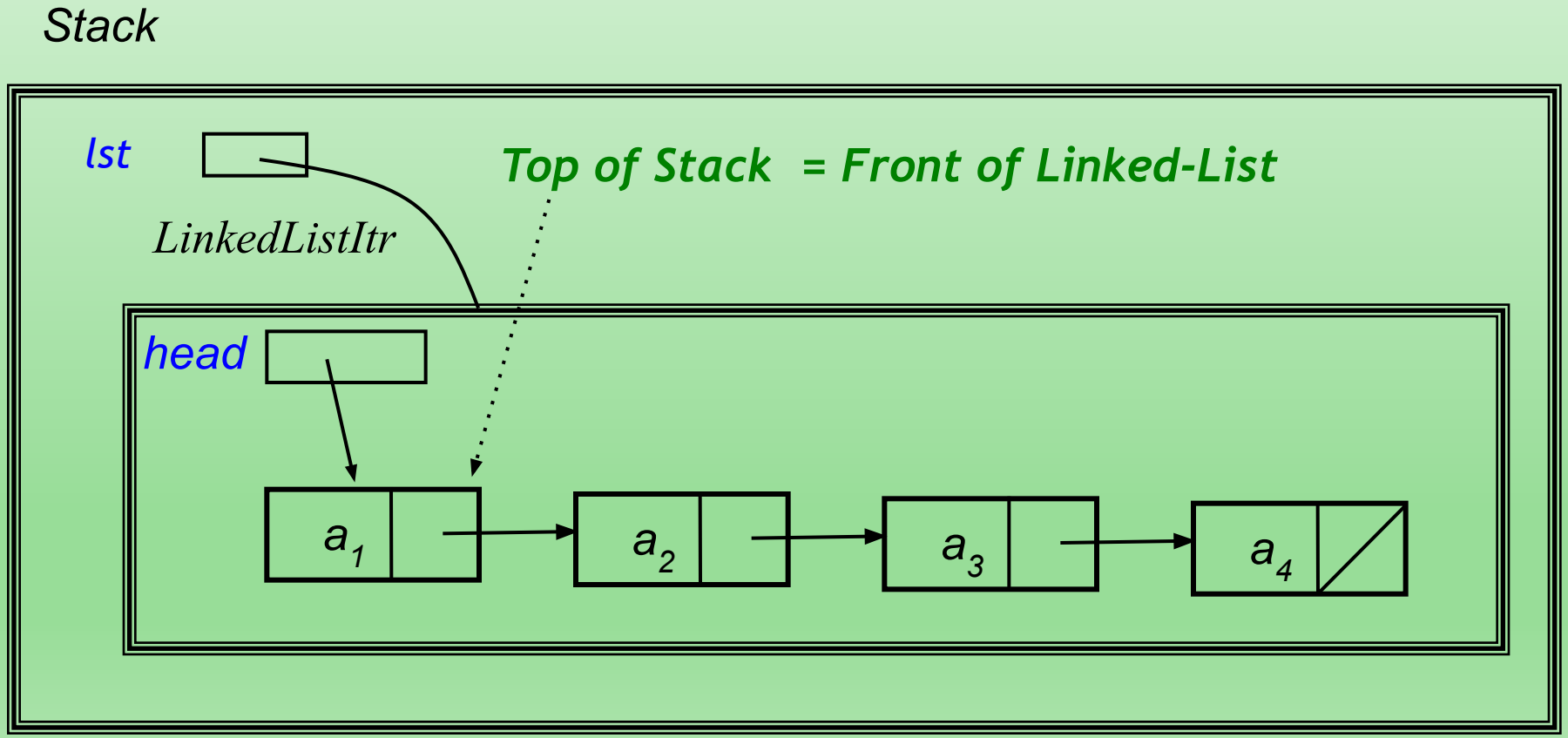
- A singly linked list is a concrete data structure consisting of a sequence of nodes
- Each node stores
  - element
  - link to the next node





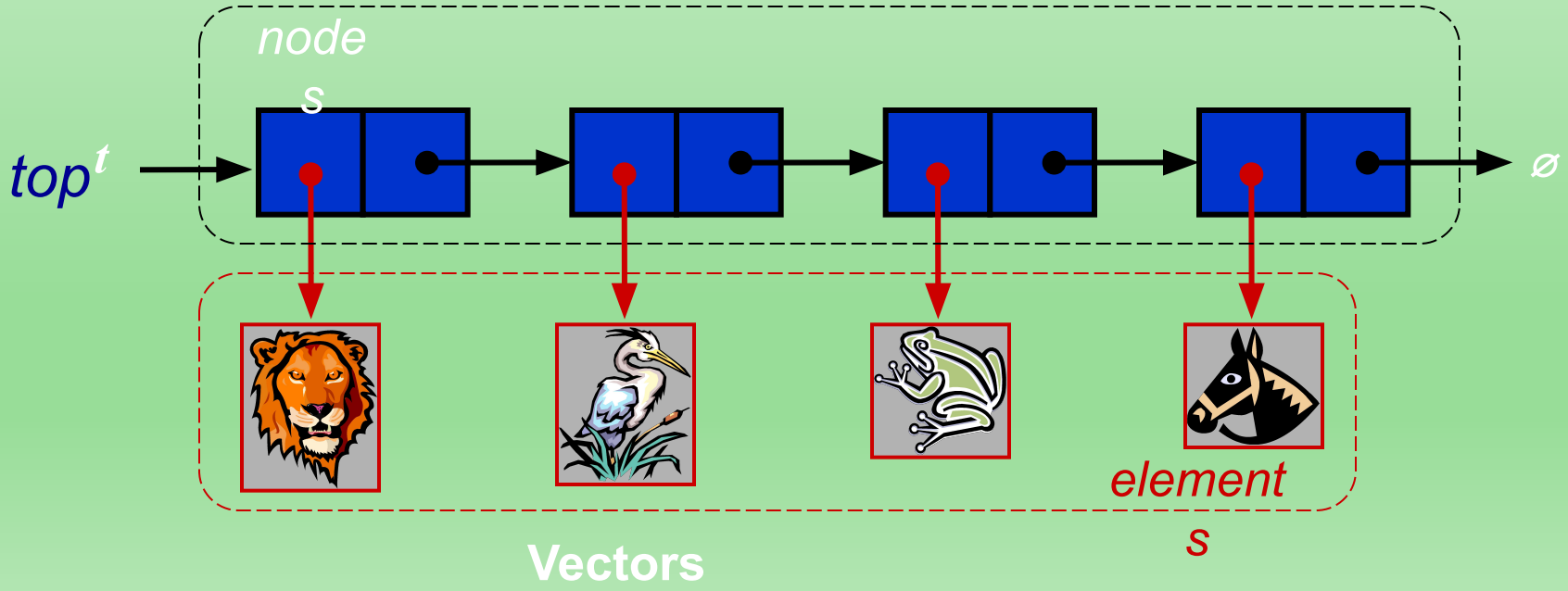
Implementation by Linked Lists

- Can use a Linked List as implementation of stack



Stack with a Singly Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



Exercise

What will an empty stack look like after the block of operations below is executed on it?

Java

```
1  push(5)
2  push(3)
3  push(7)
4  push(pop()+1)
5  peek()
```

The stack will look like the following:

```
1  (5)
2  (5,3)
3  (5,3,7)
4  (5,3,8)
```

Thus 8 will be outputted. □

Exercise

- Write a method that reverses the contents of a stack
- Solution

Python

```
1  def reverse(stack):
2    new_stack = Stack()
3    while stack.size() != 0:
4        new_stack.push(stack.pop())
5     return new_stack
```

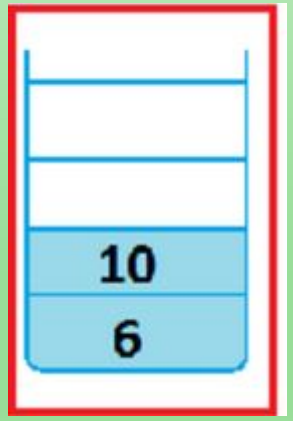
Exercise

EXAMPLE

What will an empty stack look like after the block of operations below is executed?

Java

```
1  push(6)
2  push(5)
3  push(pop()*2)
```



Stack ADT

Time Complexity

The time complexity of stacks depends on the type of the data structure you are using and the specific implementation you build. Below is a description of how linked lists and arrays are not that different when it comes to complexity:

	Linked List	Array
Push	$O(1)$	$O(1)$
Pop	$O(1)$	$O(1)$

Stack Summary

- Stack Operation Complexity for Different

	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked
Pop()	O(1)	O(1)	O(1)
Push(o)	O(1)	O(n) Worst Case O(1) Best Case O(1) Average Case	O(1)
Top()	O(1)	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)	O(1)

Applications

- Many application areas use stacks:
  - line editing
  - bracket matching
  - postfix calculation
  - function call stack
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Saving local variables when one function calls another, and this one calls another

Line Editing

- A line editor would place characters read into a buffer but may use a backspace symbol (denoted by ←) to do error correction

- *Refined Task*
  - read in a line
  - correct the errors via backspace
  - print the corrected line in reverse

Input : `abc_defgh←2klpq←←wxyz`

Corrected Input : `abc_defg2klpwxyz`

Reversed Output : `zyxwplk2gfed_cba`

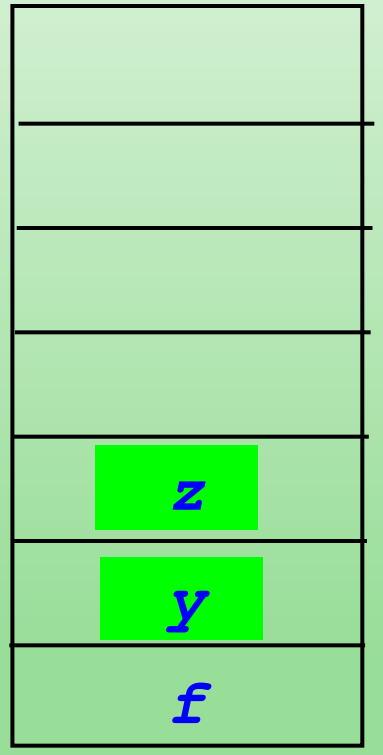
The Procedure

- Initialize a new stack
- For each character read:
  - if it is a backspace, *pop out last char entered*
  - if not a backspace, *push the char into stack*
- To print in reverse, pop out each char for output

Input : `fgh←r←←yz`

Corrected Input : `fyz`

Reversed Output : `zyf`



Stack

Bracket Matching Problem

- Ensures that pairs of brackets are properly matched
  - *An Example:* *{a,(b+f[4])*3,d+f[5]}*

  - *Bad Examples:*

(..)..) // *too many closing brackets*

(..(..) // *too many open brackets*

[..(..]..) // *mismatched brackets*

Parentheses Matching

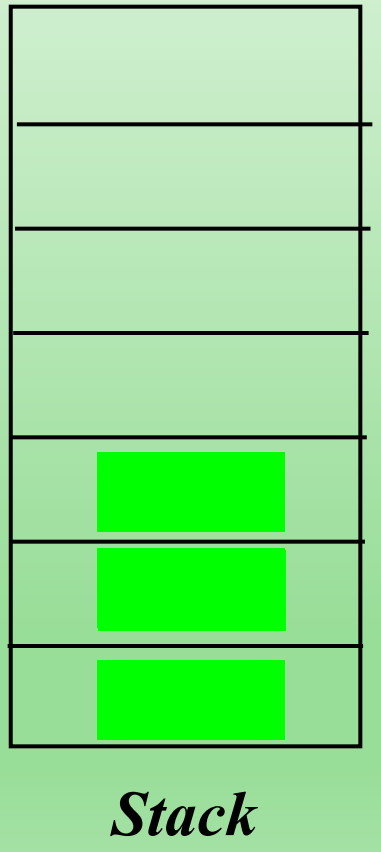
- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Informal Procedure

Initialize the stack to empty
For every char read
    if open bracket then *push onto stack*
    if close bracket, then
        return & remove most recent item
            from *the stack*
        if doesn't match then *flag error*
    if non-bracket, *skip the char read*

Example

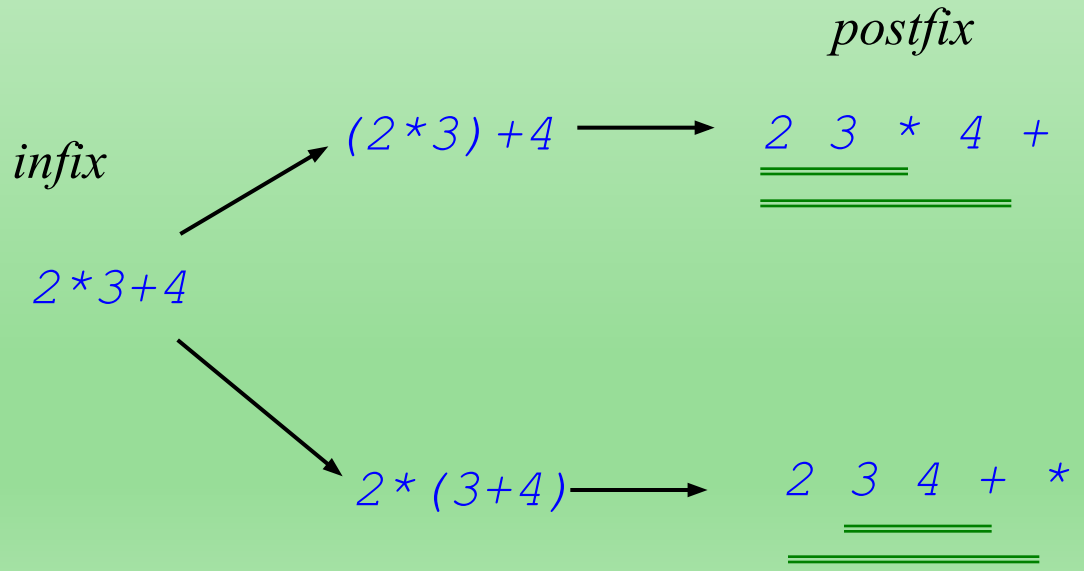
```
{a,(b+f[4])*3,d+f[5]}
```



Postfix Calculator

- Computation of arithmetic expressions can be efficiently carried out in Postfix notation with the help of a stack.

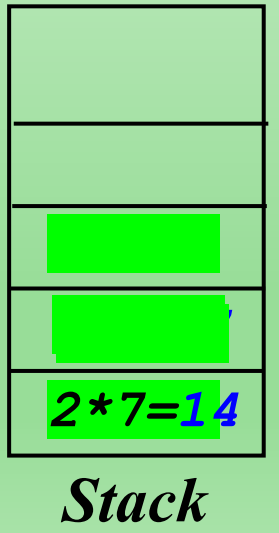
Infix - arg1 **op** arg2
Prefix - **op** arg1 arg2
Postfix - arg1 arg2 **op**



Informal Procedure

Initialise stack S
For each item read.
    If it is an operand,
        push on the stack
    If it is an operator,
        pop arguments from stack;
        perform operation;
        push result onto the stack

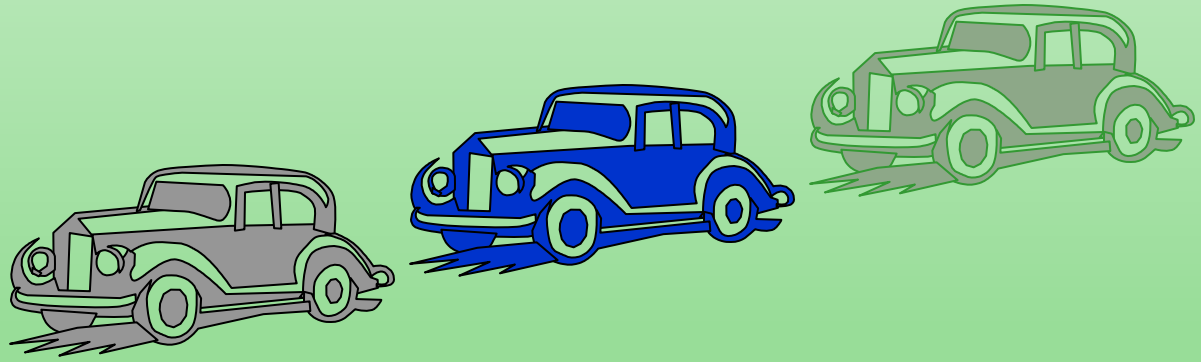
Expr	
2	*push(S, 2)*
3	*push(S, 3)*
4	*push(S, 4)*
+	*arg2=topAndPop(S)*
	arg1=topAndPop(S)
	push(S, arg1+arg2)
*	*arg2=topAndPop(S)*
	arg1=topAndPop(S)
	*push(S, arg1*arg2)*



Summary

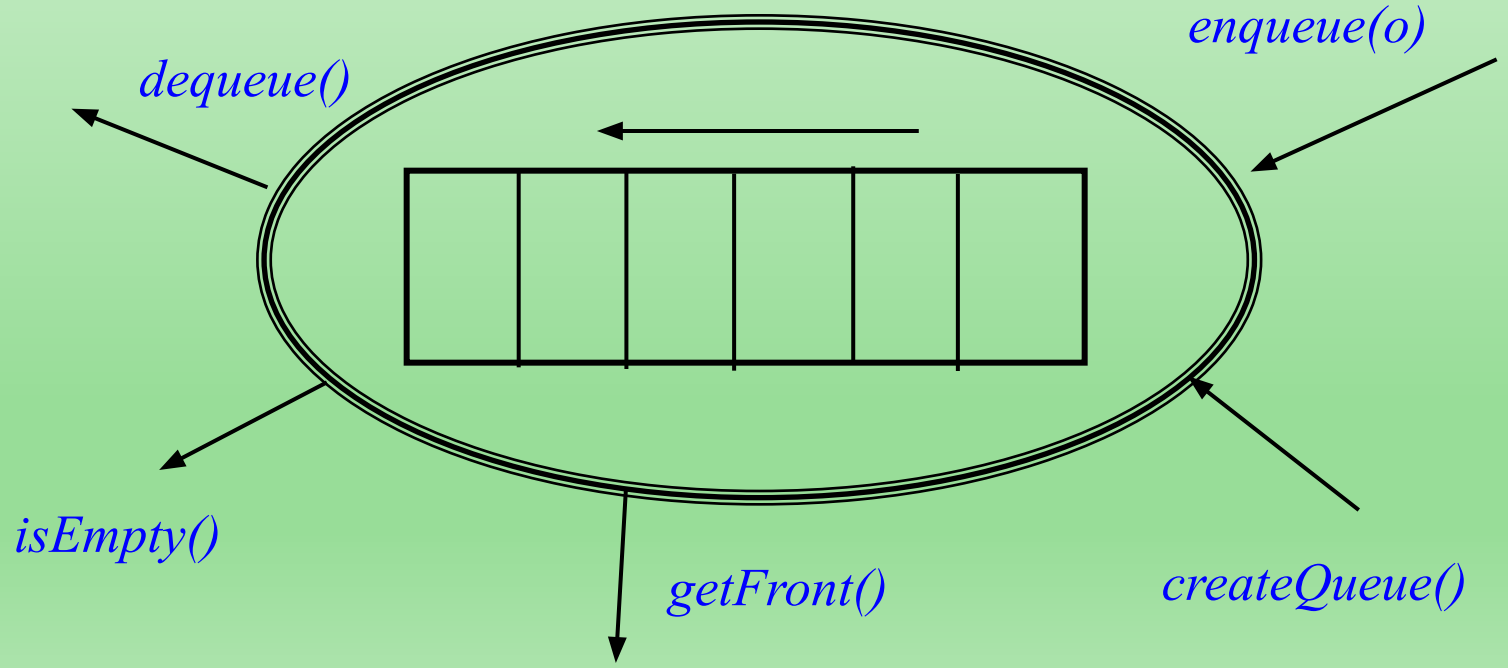
- The ADT stack operations have a last-in, first-out (LIFO) behavior
- Stack has many applications
  - algorithms that operate on algebraic expressions
  - a strong relationship between recursion and stacks exists
- Stack can be implemented using arrays or linked lists

Queues

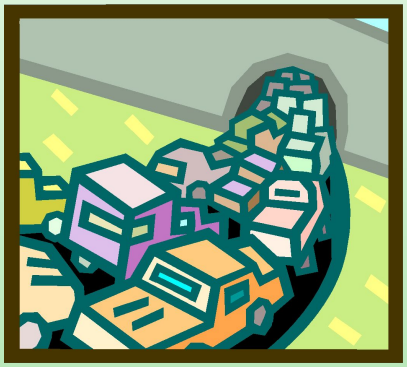


Queue ADT

- Queues implement the FIFO (first-in first-out) policy
  - An example is the printer/job queue!



Applications of Queues



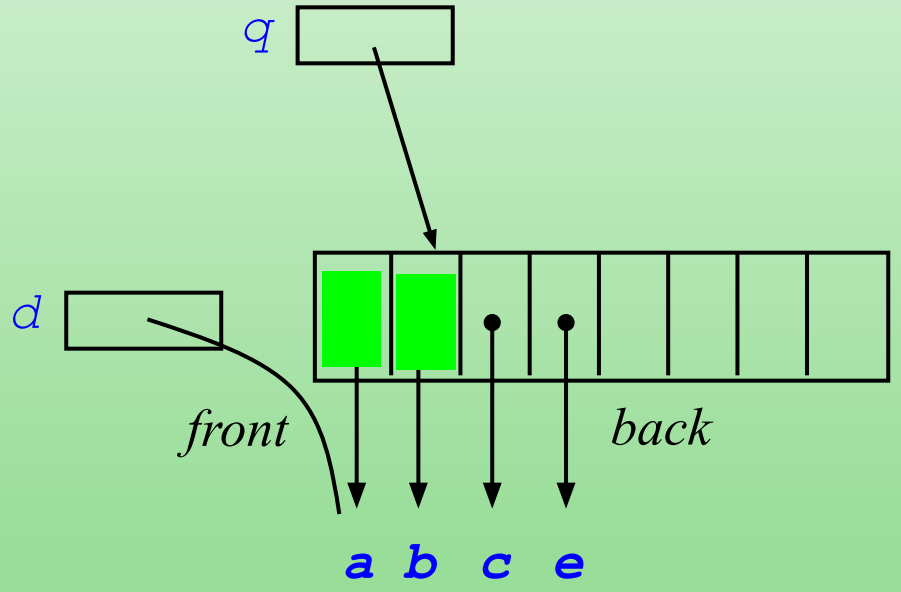
- Operating systems
- Mail services
- CPU scheduling
- Elevators
- Keyboard buffering
- Networks- The network has used a queue to store your texts so that they arrive in the same order you sent them.
- Or, when you send data to a website, you want to make sure that the data gets there in the right order.
- So, the website has a queue where it collects your data as it waits for it to be processed

The Queue ADT

- The Queue ADT stores arbitrary objects
- Insertions and deletions follow the first-in first-out (FIFO) scheme
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object o): inserts element o at the end of the queue
  - dequeue(): removes and returns the element at the front of the queue
- Auxiliary queue operations:
  - front(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a Boolean value indicating whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

Sample Operation

- *Queue *Q;*
- *enqueue(Q, “a”);*
- *enqueue(Q, “b”);*
- *enqueue(Q, “c”);*
- *d=getFront(Q);*
- *dequeue(Q);*
- *enqueue(Q, “e”);*
- *dequeue(Q);*



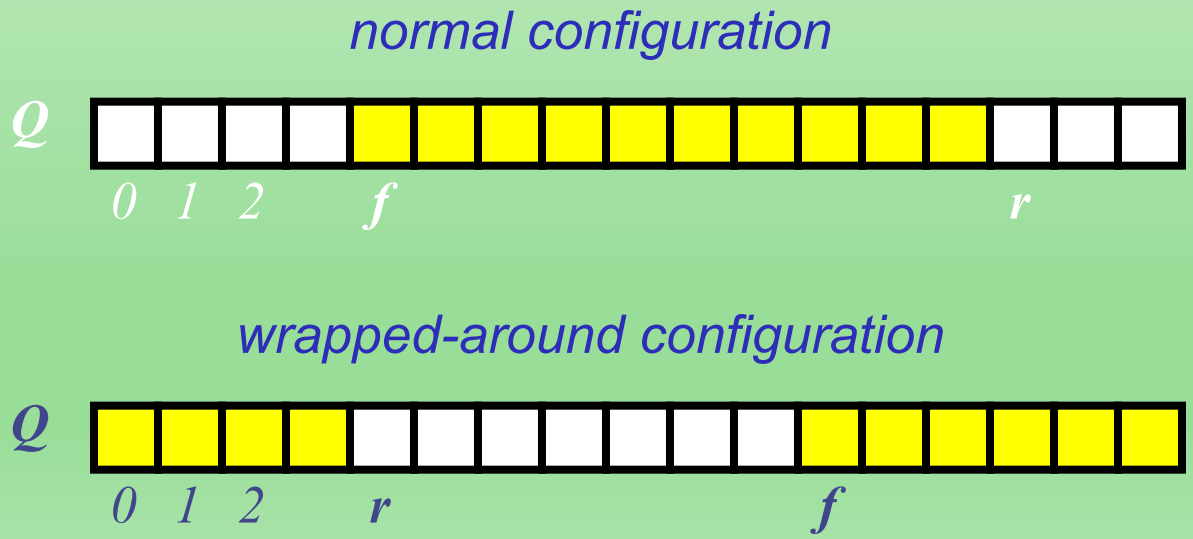
Queue ADT interface

- The main functions in the Queue ADT are (Q is the queue)

```
void enqueue(o, Q)        // insert o to back of Q

void dequeue(Q);          // remove oldest item

Item getFront(Q);         // retrieve oldest item

boolean isEmpty(Q);       // checks if Q is empty

boolean isFull(Q);      // checks if Q is full

void clear(Q);            // make Q empty
```

Array-based Queue

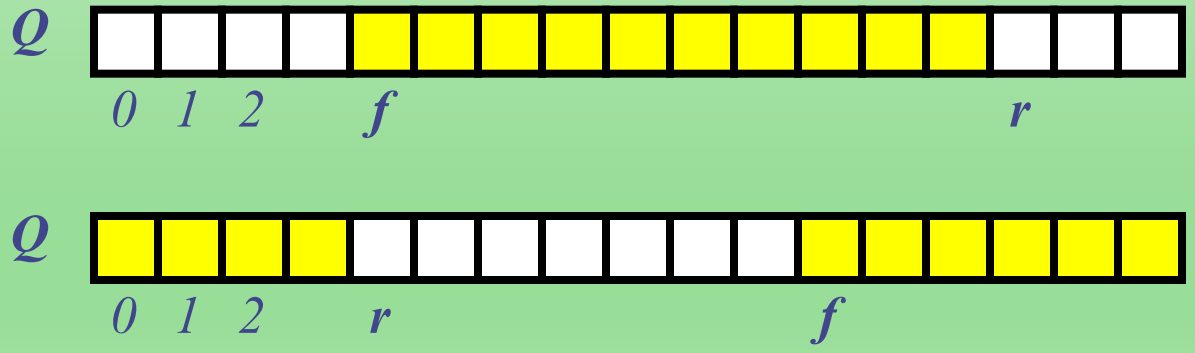
- Use an array of size ***N*** in a circular fashion
- Two variables keep track of the front and rear
  - ***f*** index of the front element
  - ***r*** index immediately past the rear element
- Array location ***r*** is kept empty



Queue Operations

- We use the modulo operator (remainder of division)

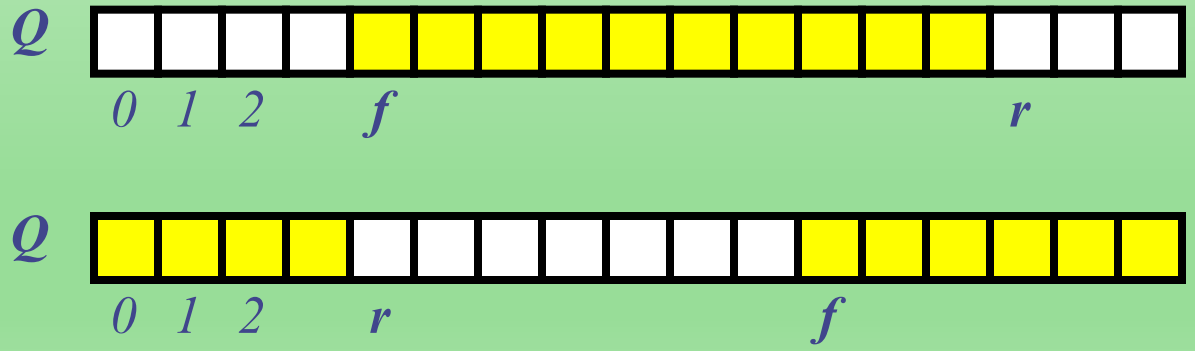
```
Algorithm size()
    return (N + r – f) mod N

Algorithm isEmpty()
    return (f = r)
```



Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

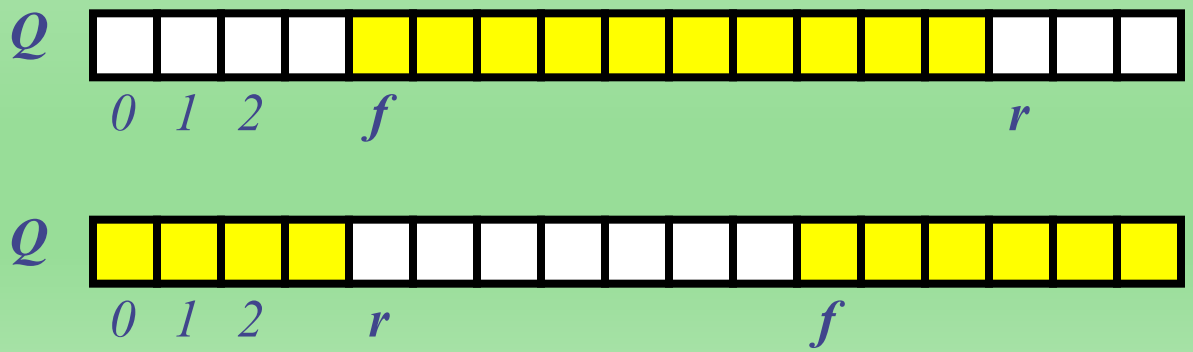
```
Algorithm enqueue(o)
    if size() = N - 1 then
        throw
FullQueueException
     else
        Q[r] = o
        r = (r + 1) mod N
```



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

```
Algorithm dequeue()
    if isEmpty() then
        throw
EmptyQueueException
    else
        o = Q[f]
        f = (f + 1) mod N
        return o
```

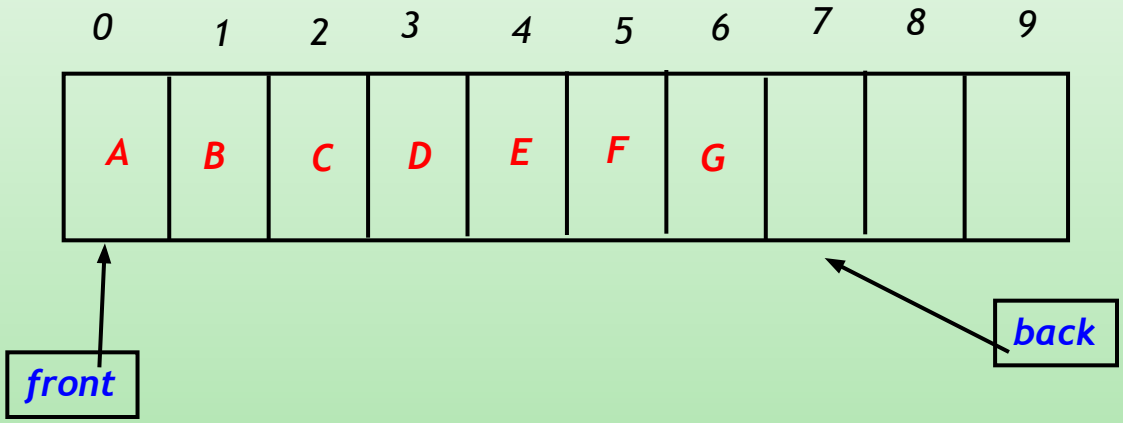


Growable Array-based Queue

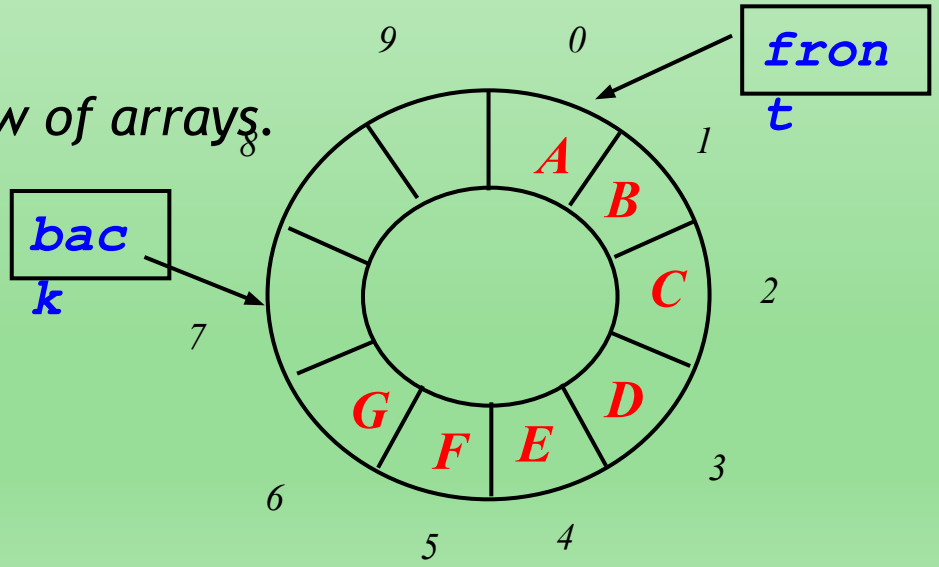
- In an enqueue operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
- Similar to what we did for an array-based stack
- The enqueue operation has amortized running time
  - $O(n)$ with the incremental strategy
  - $O(1)$ with the doubling strategy

Circular Array

- To implement queue, it is best to view arrays as circular structure



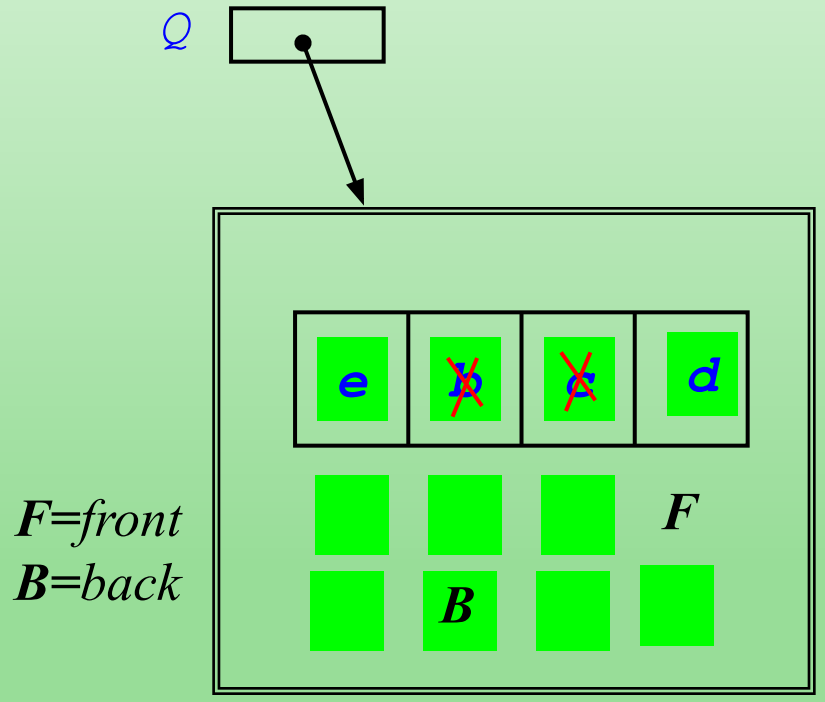
Circular view of arrays.



Sample

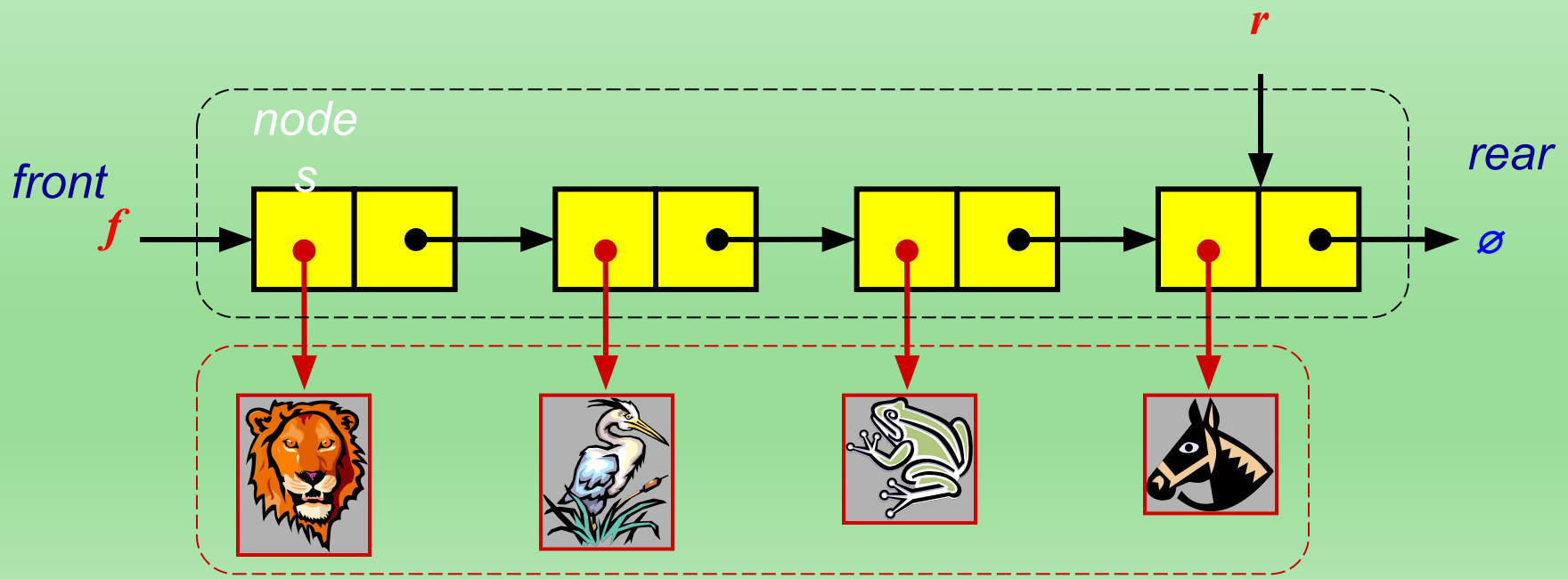
Queue *Q;

- enqueue(Q, “a”);
- enqueue(Q, “b”);
- enqueue(Q, “c”);
- dequeue(Q);
- dequeue(Q);
- enqueue(Q, “d”);
- enqueue(Q, “e”);
- dequeue(Q);



Queue with a Singly Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the head of the list
  - The rear element is stored at the tail of the list
- The space used is ***O***(***n***) and each operation of the Queue ADT takes ***O***(1) time
- NOTE: we do not have the limitation of the array based implementation on the size of the stack b/c the size of the linked list is not fixed, i.e., the queue is NEVER full.



Example: Adding and Removing Items in a Queue

The following demonstration uses the above implementation of a queue using Python.

Instantiating the queue:

```
>>> queue = Queue()
>>> queue.size()
0
```

Adding three people to the queue:

```
>>> queue.enqueue('Alice')
>>> queue.enqueue('Bob')
>>> queue.enqueue('Charlie')
>>> queue.size()
3
```

The queue currently looks like this (remember, the **head** is on the left):

```
>>> queue.qlist
['Alice', 'Bob', 'Charlie']
```

Dequeueing the queue:

```
>>> queue.dequeue()
'Alice'
>>> queue.qlist
['Bob', 'Charlie']
>>> queue.dequeue()
'Bob'
>>> queue.qlist
['Charlie']
>>> queue.dequeue()
'Charlie'
>>>queue.qlist
[]
```

Now the queue is empty and no dequeueing or peeking can be done:

```
>>> queue.dequeue()
queue underflow
>>> queue.peek()
queue is empty
```

- Consider a line of 10 people waiting to buy food at a fast-food restaurant. Imagine this line is represented by a queue.

- Where is the head of this queue, the person in front or the person in the back?

- If the person in front buys their food and leaves, how many people need to move (not including the person who left)?

- Is this O(n) time or O(1) time? Is this queue implemented using an array or a linked list?

- A1. The head of the queue is the person at the front of the line.
- A2. 9 people have to move (10 people minus the 1 who left).
- A3. This is O(n) time because it is dependent on the number of people in the line (in this case, 10). This queue is implemented using an array.

Queue Summary

- Queue Operation Complexity for Different Implementations

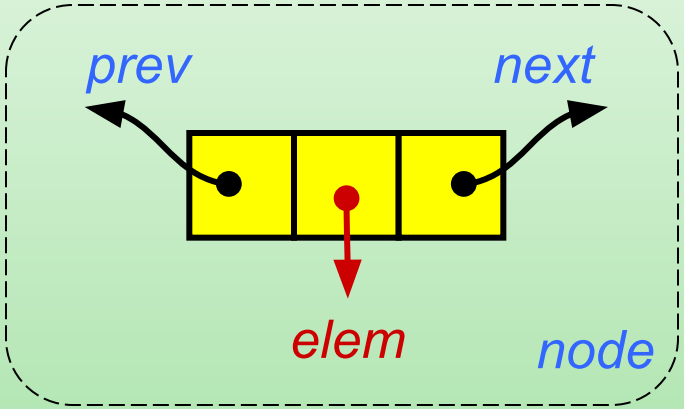
	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked
dequeue()	O(1)	O(1)	O(1)
enqueue(o)	O(1)	O(n) Worst Case O(1) Best Case O(1) Average Case	O(1)
front()	O(1)	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)	O(1)

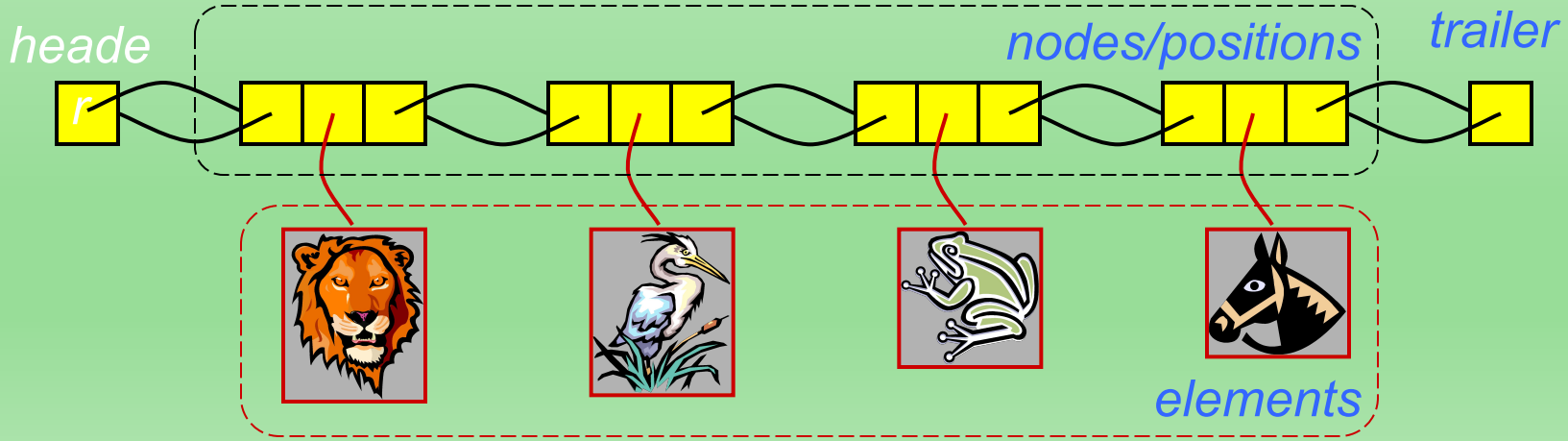
The Double-Ended Queue ADT

- The Double-Ended Queue, or Deque, ADT stores arbitrary objects. (Pronounced ‘deck’)
- Richer than stack or queue ADTs. Supports insertions and deletions at both the front and the end.
- Main deque operations:
  - insertFirst(object o): inserts element o at the beginning of the deque
  - insertLast(object o): inserts element o at the end of the deque
  - RemoveFirst(): removes and returns the element at the front of the queue
  - RemoveLast(): removes and returns the element at the end of the queue
- Auxiliary queue operations:
  - first(): returns the element at the front without removing it
  - last(): returns the element at the front without removing it
  - size(): returns the number of elements stored
  - isEmpty(): returns a Boolean value indicating whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyDequeException

Doubly Linked List

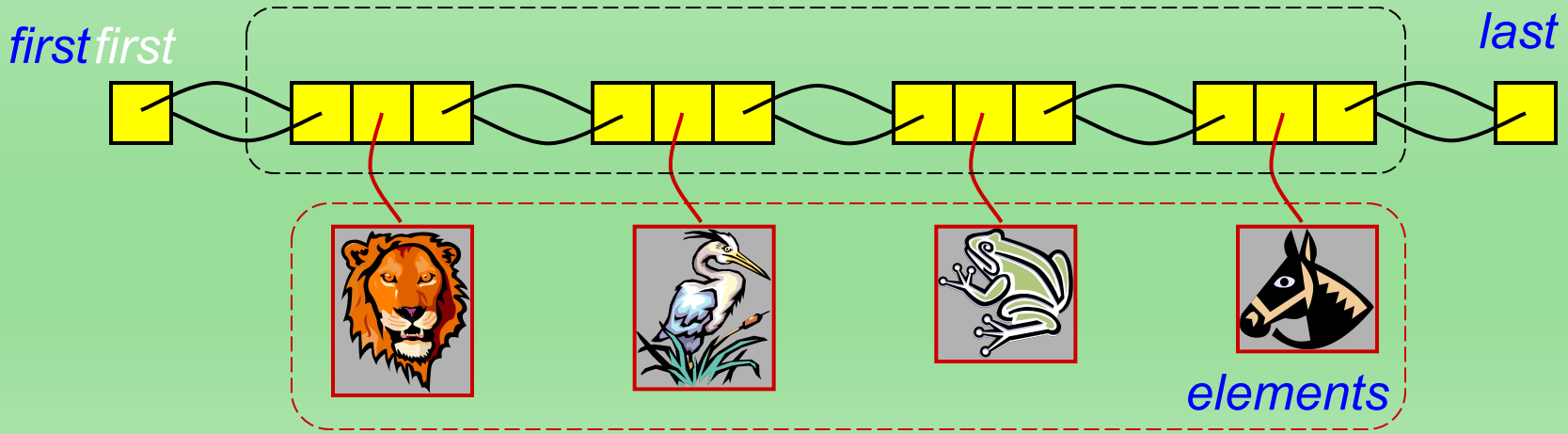
- A doubly linked list provides a natural implementation of the Deque ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes





Deque with a Doubly Linked List

- We can implement a deque with a doubly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Deque ADT takes $O(1)$ time



Performance and Limitations
- doubly linked list implementation of deque ADT

- Performance
  - Let ***n*** be the number of elements in the stack
  - The space used is ***O***(***n***)
  - Each operation runs in time ***O***(1)
- Limitations
  - NOTE: we do not have the limitation of the array based implementation on the size of the stack b/c the size of the linked list is not fixed, I.e., the deque is NEVER full.

Queue ADT

DEFINITION

There are two basic operations related to the queue:

Function Name*	Provided Functionality
`enqueue(i)`	Insert element *i* at the tail of the queue.
`dequeue()`	Remove the element at the head of the queue.

Additionally, queues are often implemented with the following operations to save computation:

Function Name*	Provided Functionality
`size()`	returns the current size of the queue
`peek()`	returns the element at the head of the queue *without changing the queue*

Deque Summary

- Deque Operation Complexity for Different

	Array Fixed-Size	Array Expandable (doubling strategy)	List Singly-Linked	List Doubly-Linked
removeFirst(), removeLast()	O(1)	O(1)	O(n) for one at list tail, O(1) for other	O(1)
insertFirst(o), InsertLast(o)	O(1)	O(n) Worst Case O(1) Best Case O(1) Average Case	O(1)	O(1)
first(), last	O(1)	O(1)	O(1)	O(1)
Size(), isEmpty()	O(1)	O(1)	O(1)	O(1)

Implementing Stacks and Queues with Deques

Stacks with Deques:

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	last()
push(e)	insertLast(e)
pop()	removeLast()

Queues with Deques:

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue()	insertLast(e)
dequeue()	removeFirst()

Disjoint Sets

Disjoint Sets

- A data structure that stores non overlapping or disjoint subset of elements is called disjoint set data structure. The disjoint set data structure supports following operations:
- Adding new sets to the disjoint set.
- Merging disjoint sets to a single disjoint set using **Union** operation.
- Finding representative of a disjoint set using **Find** operation.
- Check if two sets are disjoint or not.

- Consider a situation with a number of persons and the following tasks to be performed on them:
- Add a **new friendship relation**, i.e. a person x becomes the friend of another person y i.e adding new element to a set.
- Find whether individual **x is a friend of individual y** (direct or indirect friend)

Disjoint Set ADT

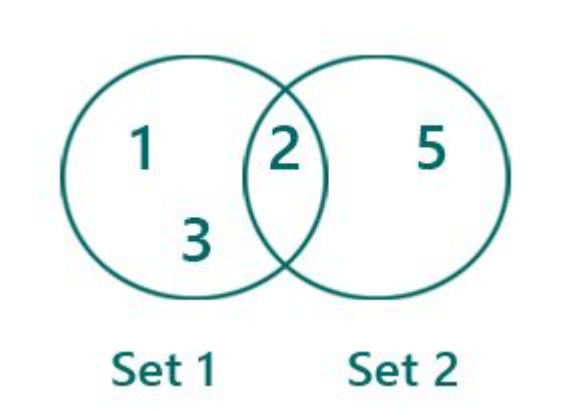
- **Disjoint sets** mean a collection of elements without any specific order.
- To implement disjoint set ADT, an array is enough also.
- ADT set is an auxiliary data structure used to solve many algorithmic problems based on graphs.

- **Disjoint Sets ADT**
- The basic operations on a disjoint set are the below ones:

1. **MAKESET(X)**: create a new set with element X
2. **UNION(X, Y)**: creating a set with X & Y and it deletes individual sets of X & Y.
3. **FIND(X)**: It finds the set in which X is there. (X can't be in two different sets since it's named as disjoint set ADT)

Disjoint Set ADT-Examples

The above is an example of disjoint set ADT whereas the below one is not since it's not disjoint.



Abstract Data Type: Disjoint Sets

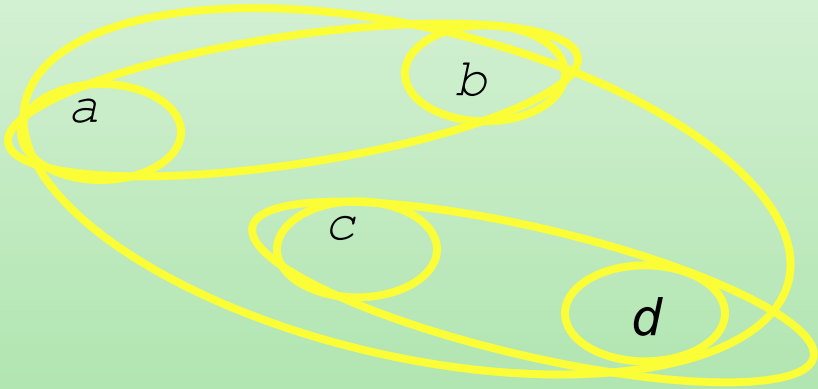
- State: collection of disjoint dynamic sets
  - The number of sets and their composition can change, but they must always be disjoint.
  - Each set has a representative element that serves as the name of the set.
  - For example, S={a,b,c} can be represented by a.
- Operations:
  - Make-Set(x): creates singleton set {x} and adds it the to collection of sets.
  - Union(x,y): replaces x's set S_x and y's set S_y with $S_x \cup S_y$
  - Find-Set(x): returns (a pointer to) the representative of the set containing x

Disjoint Sets Example

- Make-Set(a)
- Make-Set(b)
- Make-Set(c)
- Make-Set(d)
- Union(a,b)
- Union(c,d)
- Find-Set(b)
- Find-Set(d)
- Union(b,d)

returns a

returns c



- MAKESET(X), MAKESET(Y) creates two disjoint sets having element X & Y.
  Like below:

FIND(X): It returns name of the set where X exists which is here set1

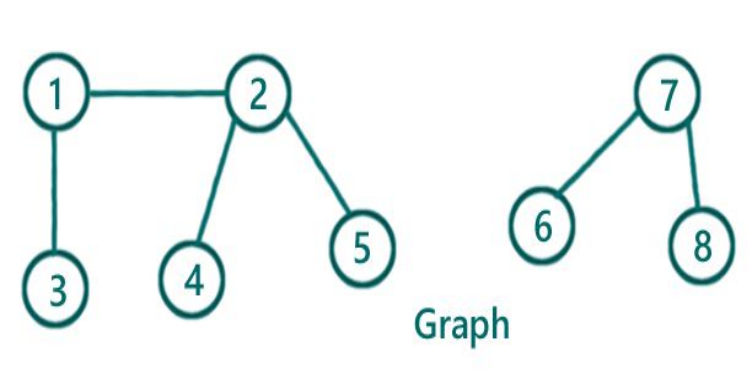
- UNION(X, Y): It creates a new set with element X & Y and removes the old sets of X & Y. UNION(X, Y) returns the below:

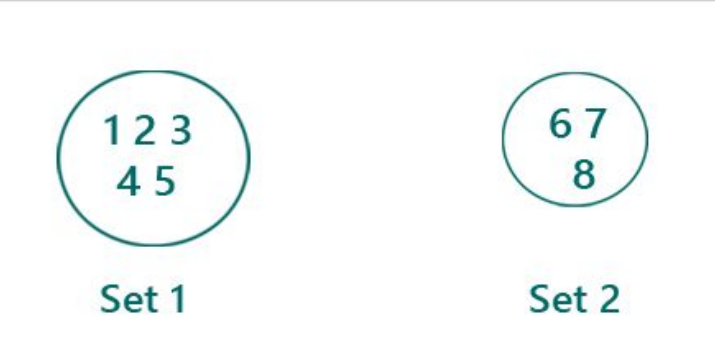
- **Application of disjoint set ADT**

1. Representing network connectivity
2. Finding cycle in an undirected graph
3. Kruskal's Minimum spanning tree algorithm
4. In gaming algorithms'

Example

- Find disjoint sets in a graph using disjoint set ADT operations FIND, UNION
- What does disjoint sets in a graph mean?
- Disjoint sets in a graph mean components of a graph. Each connected component is treated as a disjoint set since it has no relation with the other components. Each connection (edge) is said to be the relation between two nodes. Two nodes having a relation falls in the same set.
- In the below input graph-We have two components and thus two disjoint sets. Those are,
-





Example

The Disjoint Sets data structure can be used in Kruskal's MST algorithm to test whether adding an edge to a set of edges would cause a cycle.

Recall that in Kruskal's algorithm the edges chosen so far form a forest. The basic idea is to represent each connected component of this forest by its set of vertices.

In other words, each dynamic set represents a tree.

Example (continued)

Initially, form the singleton sets {v1}, …, {vn}.

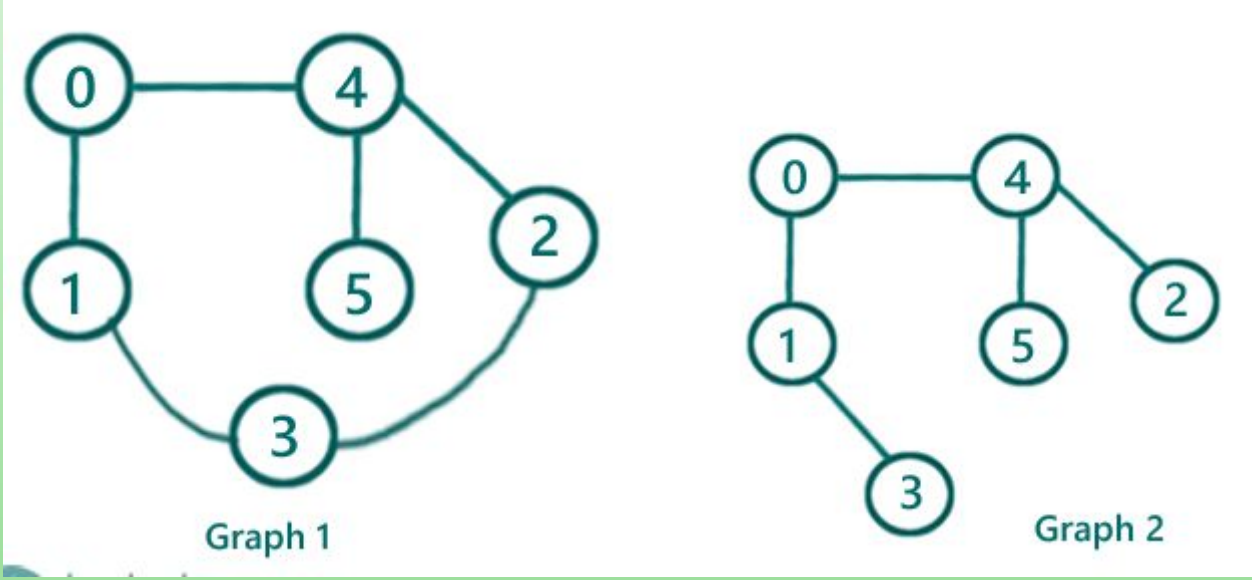
- **For all** vertices v in V **do**
  Make-Set(v).

Add an edge unless this would form a cycle:

2. **For all** edges e=(u,v) taken in increasing weight **do**
   Union(u,v) **unless** Find-Set(u)=Find-Set(v).

Example- **Finding a cycle in an undirected graph using Disjoint Sets**

- In the below example, graph 1 has a cycle where graph2 don't have any cycle.

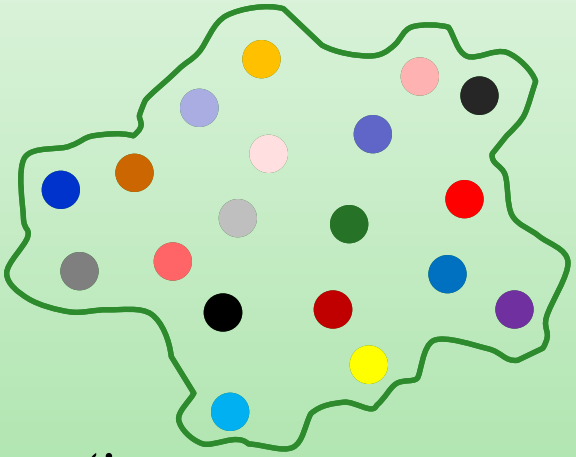


How we can use disjoint set ADT to find whether there is a cycle or not.

- An edge is a relation b/w two nodes and two nodes having an edge b/w them, are supposed to be in the same disjoint set.
- Through, ADT operation FIND(X) & UNION(X, Y), we create disjoint sets out of the available edges.
- To detect a cycle in an undirected graph the algorithm would be:
- For each edge in the edge list:
- Find parents(set name) of the source and destination nodes respectively (Though we are using terms like source & destination node, the edges are undirected).
- If both have the same parent, that means they belong to the same set already and this edge creates a cycle. Since both the node already belong to the same set, that means even without this edge we could have reached one node from other. So, the addition of this edge creates a cycle since edges are undirected.
- Else
- Do the UNION of them to bring them under the same set.

A Key Idea

If you put marbles into a sack of marbles, how do you get back your *original* marbles?



You only can do that if all marbles are somehow unique.

The Dictionary and Set ADTs insist that everything put inside of them must be unique (i.e., no duplicates).

This is achieved through keys.

The Dictionary (a.k.a. Map) ADT

Data:

- Set of (key, value) pairs
- keys are mapped to values
- keys must be comparable
- keys must be unique

Standard Operations:

- insert(key, value)
- find(key)
- delete(key)

Dictionary ADT

Dictionary ADT

- Dictionary (map, association list) is a data structure, which is generally an association of unique keys with some values.
- One may bind a value to a key, delete a key (and naturally an associated value) and lookup for a value by the key. Values are not required to be unique.
- Simple usage example is an explanatory dictionary. In the example, words are keys and explanations are values.

Operations

- **Dictionary create()** creates empty dictionary
  boolean isEmpty(Dictionary d) tells whether the dictionary **d** is empty
  put(Dictionary d, Key k, Value v) associates key **k** with a value **v;** if key **k** already presents in the dictionary old value is replaced by **v**
  Value get(Dictionary d, Key k) returns a value, associated with key **k** or null, if dictionary contains no such key
  remove(Dictionary d, Key k) removes key **k** and associated value
  destroy(Dictionary d) destroys dictionary **d**

Dictionaries

Dictionaries are often implemented as **hash tables**

Here a dictionary has been declared and initialized.
The identifier (name) of the dictionary is results.
The key is the name of the student and the value is the result of a recent assessment.

```
DICTIONARY results = {"Detra" : 17,
            "Nova" : 84,
            "Charlie" : 22,
            "Hwa" : 75,
            "Roxann" : 92,
            "Elsa" : 29}
```

The use of a student's first name as a key is unlikely to be a sensible choice as you would probably have more than one student with the same name (and key values must be unique).
Note that a dictionary is **unordered**.
No attempt should be made to keep the data in any kind of order, as it is arranged to optimise the speed of retrieving a value by key rather than by position.

Retrieve a value from a dictionary

- To retrieve a value from a dictionary you must use the key. This will return the associated data value.
- The key is used in the same way that an index is used for an array.
- In pseudocode and in many programming languages, the key value (or a variable containing the key value) is enclosed within square brackets.

```
1 score = results["Charlie"]
2 print(score)
```

- A procedure based on the example above will display 22, which is the value associated with the key **Charlie**.
- Depending on the programming language, attempting to retrieve data using a missing key may give a default value or throw an exception.

Examples

1. Given the contents of the following dictionary named scores:
DICTIONARY scores = {13 : 23, 9 : 84, 46 : 22, 23 : 75}

What value will be retrieved if you specify scores[23]?

2. DICTIONARY countries = {"France" : ["Paris", 67], "Germany" : ["Berlin", 83], "Belgium" : ["Brussels", 23], "Ireland" : ["Dublin", 11]}

What does countries["Ireland"] return?

Answers

- **Answer: 75**
- The contents of the dictionary are key-value pairs. The first item in the dictionary has the key 13 and the next key is 9. Key 23 is associated with the value 75.

Answer :
The values in the dictionary countries are arrays. Therefore, the statement: countries["Ireland"] will return ["Dublin", 11]

Each array has two elements. The first is in position 0,0 and the second is in position 1,1.

Therefore countries["Ireland"][1] will return the second element from the array ["Dublin", 11] which is 11.

Insert a new value

- To insert a new value you need to provide a key and the data. Remember that the key must be unique; if the key does not exist in the dictionary, the key-value pair is inserted.
- Since the key "Bob" does not currently exist in the dictionary, a new entry will be made for the key Bob with the associated value 78.

```
1 results["Bob"] = 78
```

Update a value in the dictionary

To update a value in the dictionary, you need to provide a valid key and the associated data. If the key does not exist in the dictionary, then a new key-value pair will be inserted instead.

```
1 results["Hwa"] = 71
```

*Here the value associated with the key **Hwa** will be updated to 71 (from 75).*

Remove a key-value pair

- If you remove a key, its associated value will be removed as well. If the key does not exist in the dictionary,
- an exception will be thrown.

```
1 results.pop("Elsa")
```

The Set ADT

Data:

- keys must be comparable
- keys must be unique

Standard Operations:

- insert(key)
- find(key)
- delete(key)

Comparing Set and Dictionary

Set and Dictionary are essentially the same

- Set has no values and only keys
- Dictionary's values are "just along for the ride"
- The same data structure ideas thus work for both dictionaries and sets
- We will thus focus on implementing dictionaries

But this may not hold if your Set ADT has other important mathematical set operations

- Examples: union, intersection, isSubset, etc.
- These are binary operators on sets
- There are better data structures for these

Sets as ADT

- Sets are a type of abstract data type that allows you to store a list of non-repeated values. Their name derives from the mathematical concept of finite sets.

- Unlike an array, sets are unordered and unindexed.

- You can think about sets as a room full of people you know. They can move around the room, changing order, without altering the set of people in that room. Plus, there are no duplicate people (unless you know someone who has cloned themselves).

- These are the two properties of a set: the data is unordered and it is not duplicated.

SET ADT

The set has four basic operations.

Function Name*	Provided Functionality
`insert(i)`	Adds *i* to the set
`remove(i)`	Removes *i* from the set
`size()`	Returns the size of the set
`contains(i)`	Returns whether or not the set contains *i*

Sometimes, operations are implemented that allow interactions between two sets

Function Name*	Provided Functionality
`union(S, T)`	Returns the union of set *S* and set *T*
`intersection(S, T)`	Returns the intersection of set *S* and set *T*
`difference(S, T)`	Returns the difference of set *S* and set *T*
`subset(S, T)`	Returns whether or not set *S* is a subset of set *T*

Examples

1) What will be the output of the following?

```
Python
1  >>> set1 = Set([1, 2, 3, 4, 1, 4])
2  >>> set1
```

2) What will be the output of the following?

```
Python
1  >>> set1 = Set([1, 3, 5, 7])
2  >>> set2 = Set([2, 3, 4, 5])
3  >>> set1.union(set2)
```

3) What will be the output of the following?

```
Python
1  >>> set1 = Set([1, 3, 5, 7])
2  >>> set2 = Set([2, 3, 4, 5])
3  >>> set1.intersection(set2)
```

4) What will be the output of the following?

```
Python
1  >>> set1 = Set([1, 2, 3])
2  >>> set2 = Set([2, 3, 4])
3  >>> set1.subset(set2)
```

Answers

This set was instantiated with multiple values of 1 and 4. So, those will be removed

```
Python
1  >>> set
2  [1, 2, 3, 4]
```

We're taking the union of both sets, so we want each element from both sets, but we exclude duplicate because we're using sets.

```
Python
1  >>> set1.union(set2)
2  [1, 2, 3, 4, 5, 7]
```

To take the intersection from both sets, we want to return one copy of each element that exist in *both* sets.

```
Python
1  >>> set1.intersection(set2)
2  [3, 5]
```

We're asking if *set1* is a subset of *set2*. *Set1* has the value 1 in it, and *set2* doesn't. So, it is false.

```
Python
1  >>> set1.subset(set2)
2  False
```

Time Complexity

- Hash Table Insert - O(1)
- Hash Table Remove - O(1)
- Hash Table Contains - O(1)

Uses

Any time you want to store information according to some key and then be able to retrieve it efficiently, a dictionary helps:

- Networks: router tables
- Operating systems: page tables
- Compilers: symbol tables
- Databases: dictionaries
- Search: inverted indexes, phone directories, …

Dictionary: ADTs

Arrays and linked lists are viable options, just not great good ones.

For a dictionary with n key/value pairs, the worst-case performances are:

	Insert	Find	Delete
Unsorted Array	O(1)	O(n)	O(n)
Unsorted Linked List	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(log n)	O(n)
Sorted Linked List	O(n)	O(n)	O(n)

Again, the array shifting is costly

Trees

Tree Data Structure

Introduction

A tree is recursively defined non-linear (hierarchical) data structure. It comprises nodes linked together in a hierarchical manner. Each node has a label and the references to the child nodes. Figure 1 shows an example of a tree.

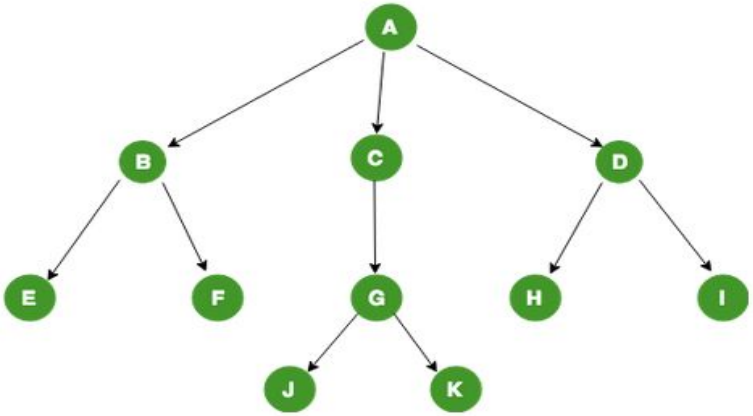


Fig 1: An example of a tree

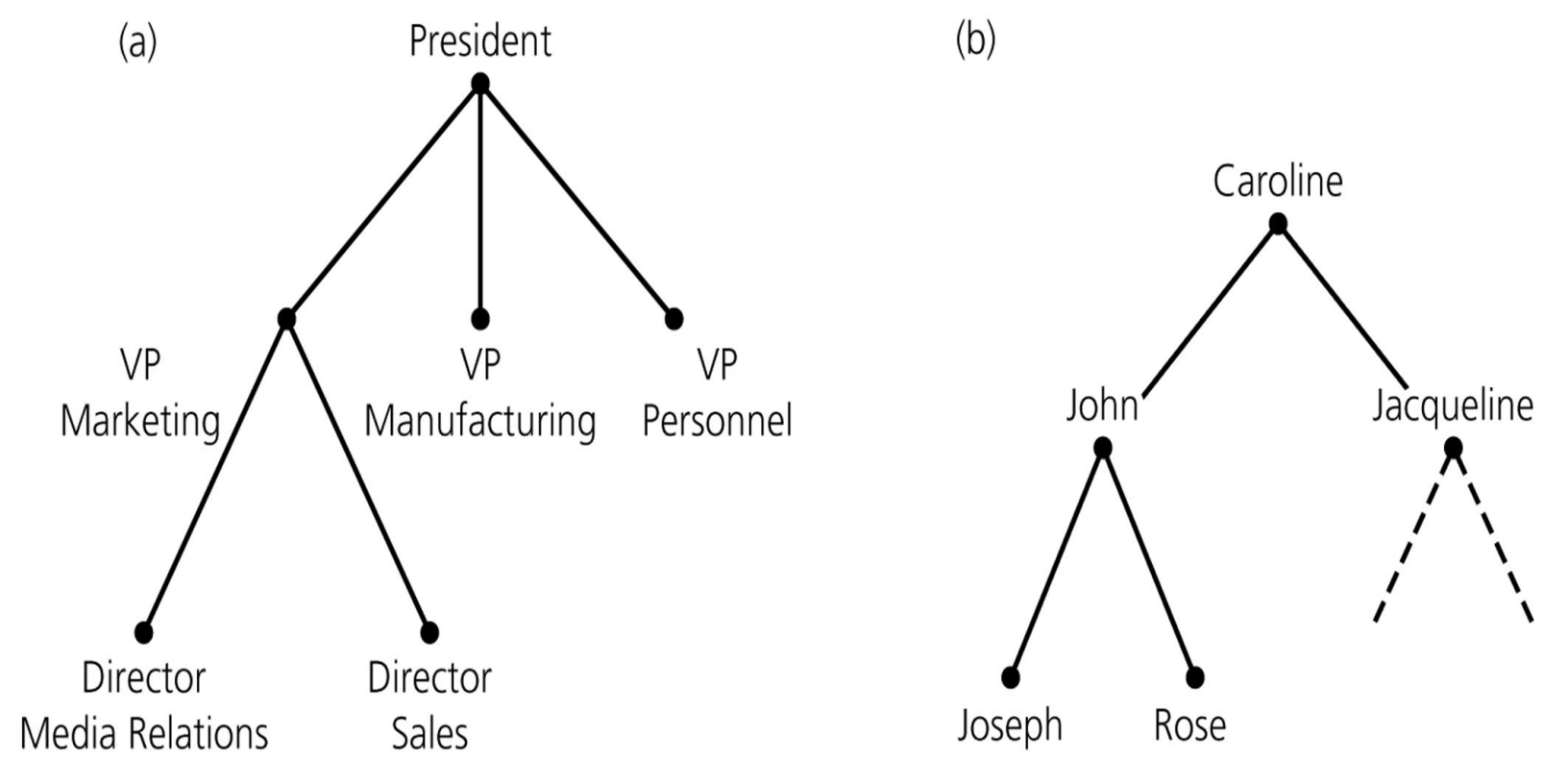
One of the nodes in the tree is distinguished as a *root* and we call this tree as a *rooted tree*. In figure 1, the node labeled A is a root of the tree. All the nodes except the root node have exactly one parent node and each node has 0 or more child nodes.

Mathematically, a tree can be defined as an *acyclic*, *undirected*, and *connected* graph.

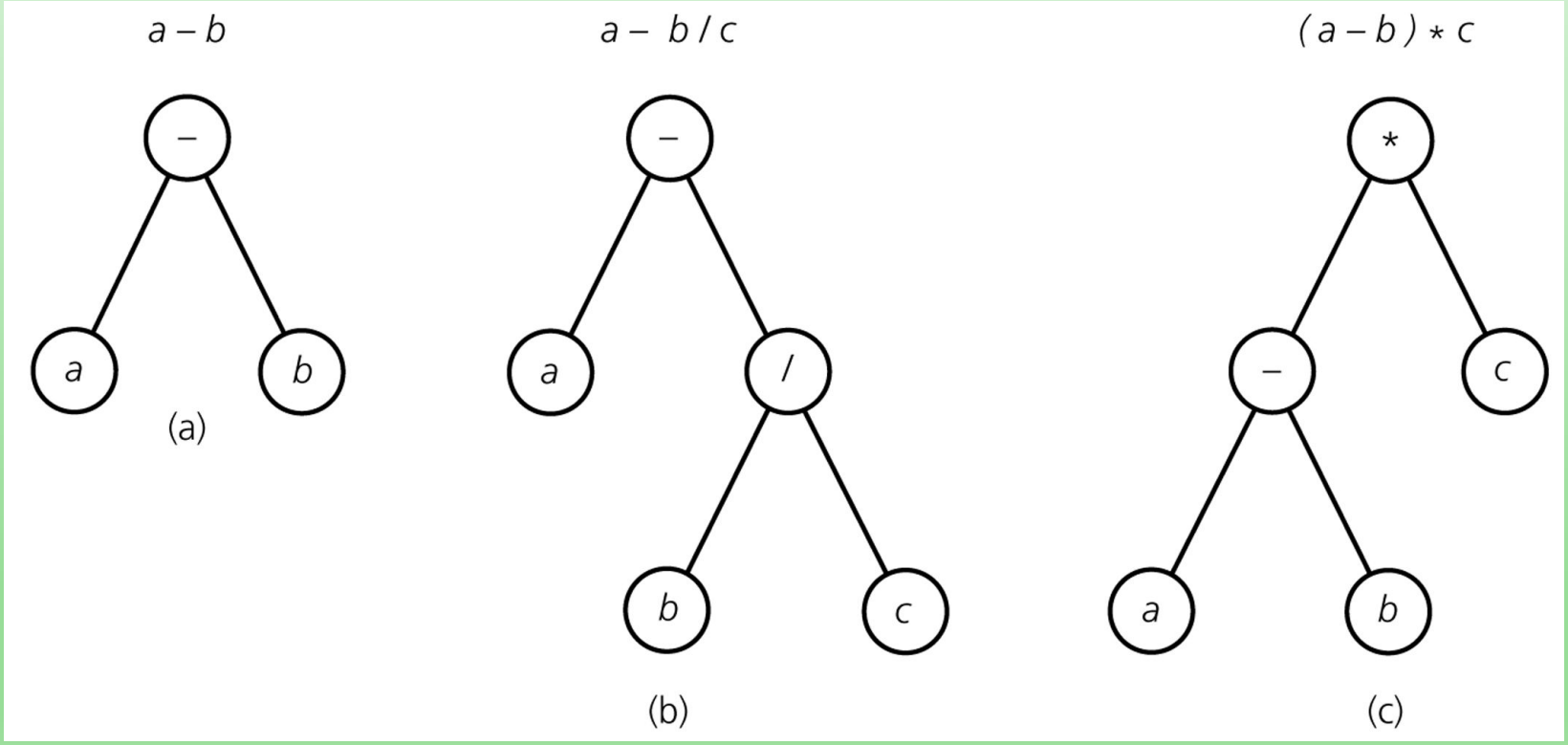
- **Acyclic**: There are no cycles i.e. there is exactly one path that connects every node to every other node. For example, In figure 1, to go from E to D there is only one path E-B-A-D.
- **Undirected**: All the edges in the tree are bidirectional.
- **Connected**: From every node, we can go to every other node. The disconnected trees are called a forest.

In mathematics and graph theory, trees are mostly undirected but in computer science, a tree is mostly assumed to be directed. That means in the tree data structure, we don't have references that point to the parents.

General Tree v.s. Binary Tree



Represent Algebraic Expressions using Binary Tree



Applications of tree data structure

1. Trees are best to store information that forms the tree structure naturally. Examples: File System, Syntax trees

2. Storing information in trees are faster than storing information in arrays and linked lists for operations like insert or read or delete.

3. Height balanced trees like a Red-Black tree or AVL tree have O(log n) time complexity for all the operations.

4. Trees like B- tree and B+ trees are used in the database for indexing.

5. One special kind of binary tree called heap is useful in HeapSort, Priority queue etc.

Tree Data Structure

Node

A node is an element of a tree. A tree is nothing but nodes linked together. Figure 2 illustrates this.

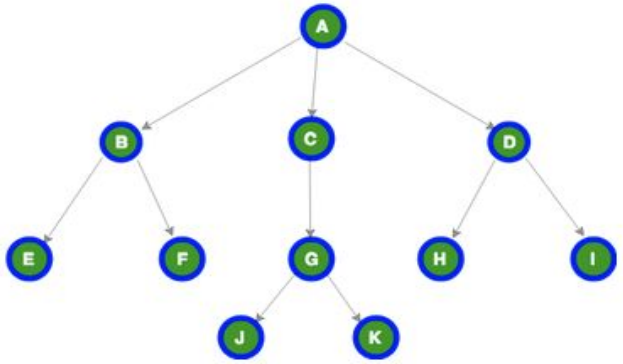


Fig 2: The nodes of the tree

All the circle in the figure are nodes.

Root

Root is the first node of the tree. We can consider root as an origin of a tree. A tree has exactly one root. Once we have the root, we can get to any nodes in the tree. Figure 3 illustrates a root node.

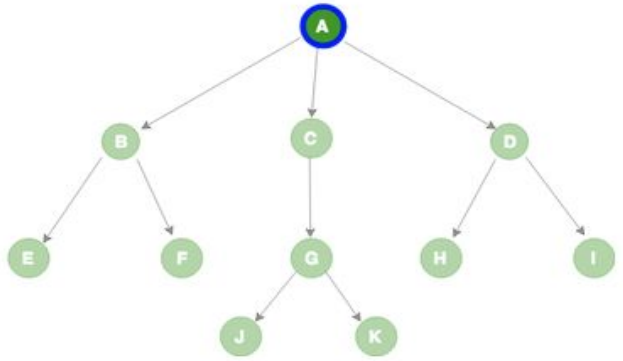


Fig 3: A root of the tree

Tree Data Structure

Child

A child is a node that is directly connected to another node when moving away from the root. In Figure 4, (B, C, D) are child nodes of A, (E, F) are child nodes of B, G is a child node of C, (H, I) are child nodes of D and (J, K) are child nodes of G.

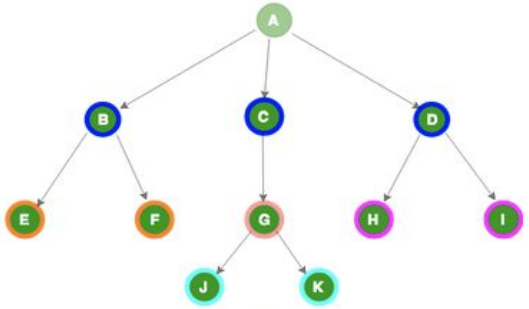


Fig 4: Child Nodes

Parent

A parent node is a node that has one or more child nodes. This is a converse notion of a child node. In Figure 5, A is a parent of (B, C, D), B is a parent of (E, F), C is a parent of G, D is a parent of (H, I) and G is a parent of (J, K).

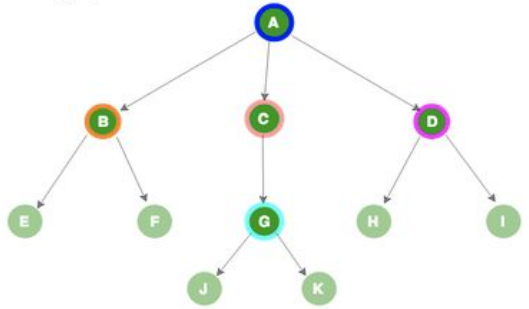


Fig 5: Parent Nodes

Siblings

Tree Data Structure

Siblings

A group of nodes with the same parent are called siblings. Figure 6 shows the siblings in the tree. The nodes with the same border color are siblings.

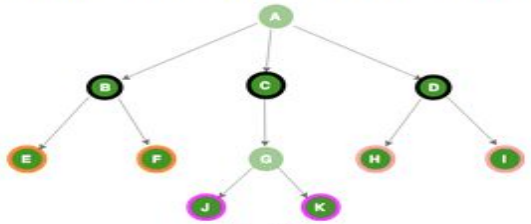


Fig 6: Siblings

Leaf Node

A node with no children is called a leaf. A leaf node is also called an external node or a terminal node. We can not go further down the tree from the leaf node. In Figure 7, E, F, J, K, H and I are leaf nodes.

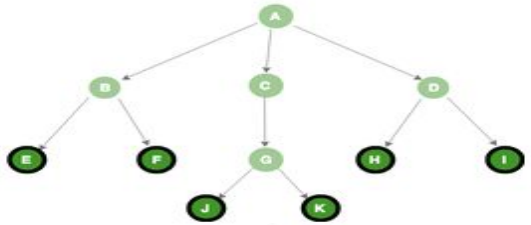


Fig 7: Leaf Nodes

Internal Node

An internal node has at least one child. All nodes except leaf nodes are internal nodes of the tree. It is also called a branch node. Figure 8 shows the internal nodes of the tree.

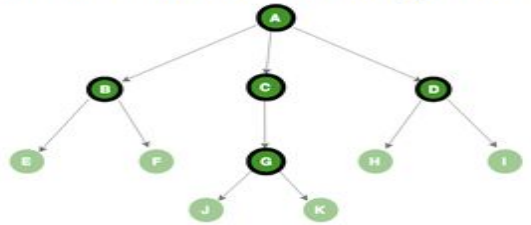


Fig 8: Internal Nodes

Edge

An edge is a connection between one node and another. In a tree, parent and child are connected by an edge.

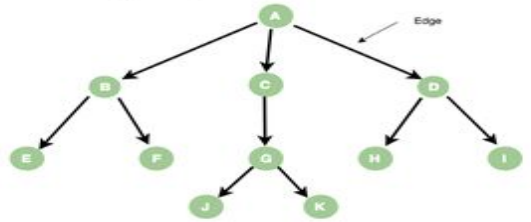


Fig 9: Visualization of edges of a tree

Descendant

A descendant node of a node is any node in the path from that node to the leaf node (including the leaf node). The immediate descendant of a node is the "child" node. In figure 10, the descendant nodes of node C are G, J and K

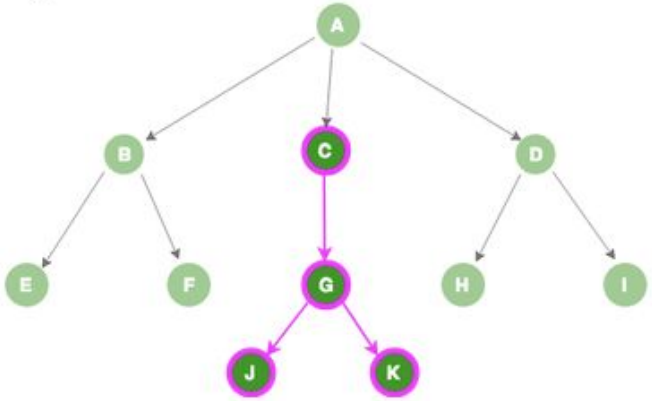


Fig 10: Descendants of node C

Ancestor

An ancestor node of a node is any node in the path from that node to the root node (including the root node). In figure 10, the ancestor nodes of node J are G, C and A.

Path

The sequence of nodes and edges connecting a node and its descendants is called a path. In Figure 11, the path between A and E is A-B-E and the path between C and J is C-G-J

]

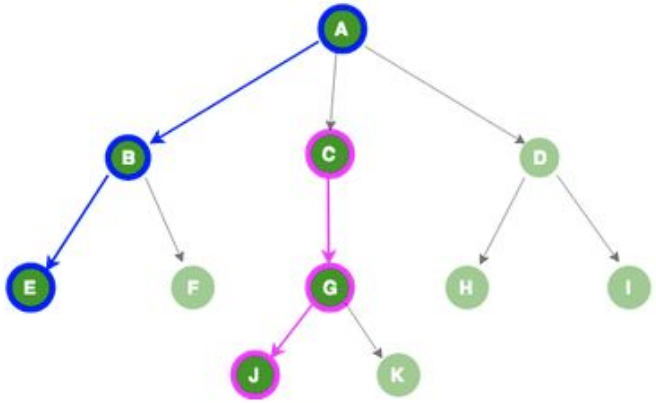


Fig 11: Illustrating Paths

Ancestor

An ancestor node of a node is any node in the path from that node to the root node (including the root node). In figure 10, the ancestor nodes of node J are G, C and A.

Path

The sequence of nodes and edges connecting a node and its descendants is called a path. In Figure 11, the path between A and E is A-B-E and the path between C and J is C-G-J

]

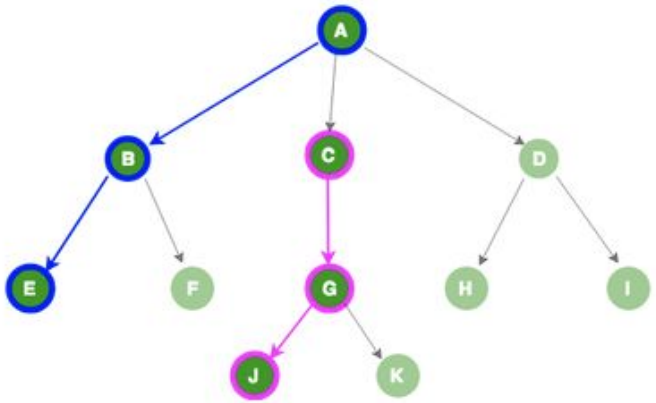


Fig 11: Illustrating Paths

Degree

A degree of a node is the number of children. Since leaf node doesn't have children, its degree is 0. The degree of a node can be any whole number from 0 to N. Figure 12 shows nodes with the corresponding degree.

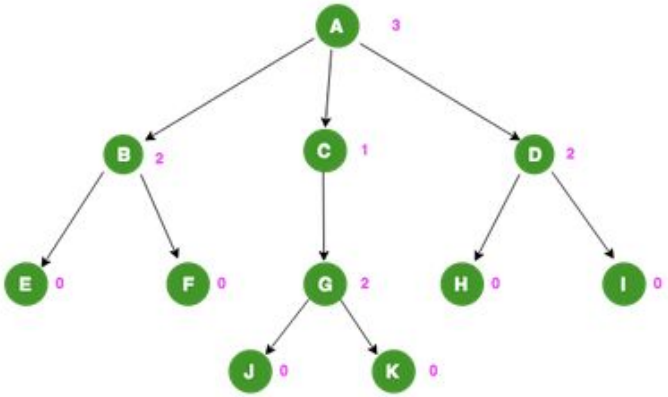


Fig 12: A tree with nodes and corresponding degree

Level

A level of a particular node is the number of edges between the node and a root + 1. The level always starts with 1 i.e. the level of a root is 1. The children of the root have level 2. The children of a level 2 node have level 3 and so on. Figure 13 illustrates this.

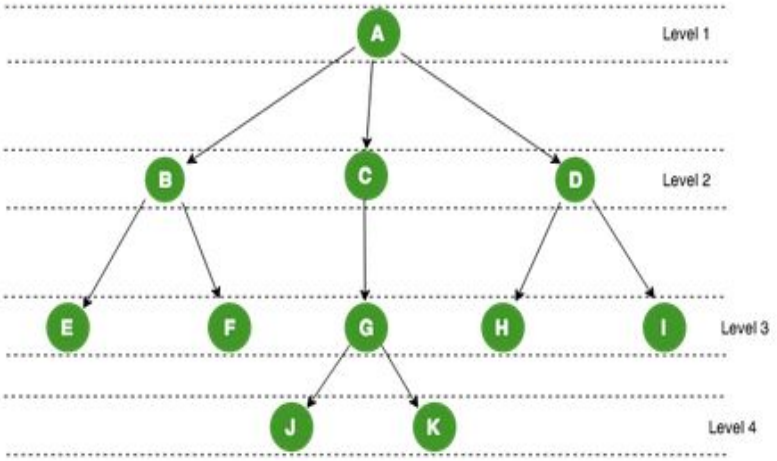


Fig 13: Level of nodes

Height

The height of a node is the number of edges on the longest path between that node and a leaf. The height of a leaf node is 0 and the root has the largest height. The height of the root node is called the *Height of Tree*. In Figure 14, the height of the tree is 3.

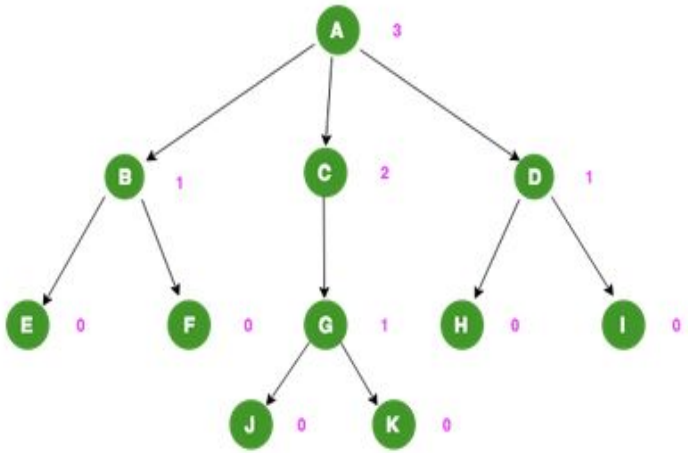


Fig 14: Height of nodes

Depth

The depth of a node is the number of edges between the node and the root of the tree. The depth of the tree is 0. The depth of the tree is the depth of a leaf node on the longest path. In Figure 15, the depth of the tree is 3.

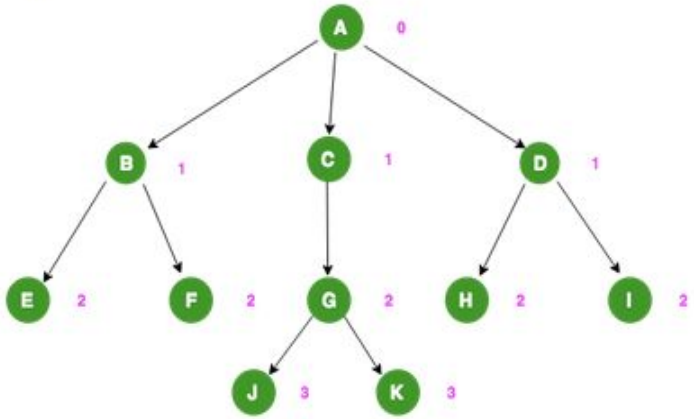


Fig 15: Depth of nodes

Subtrees

A node and all its descendants form a subtree. The subtree of a root node is the tree itself. Figure 16 shows some of the subtrees of the tree.

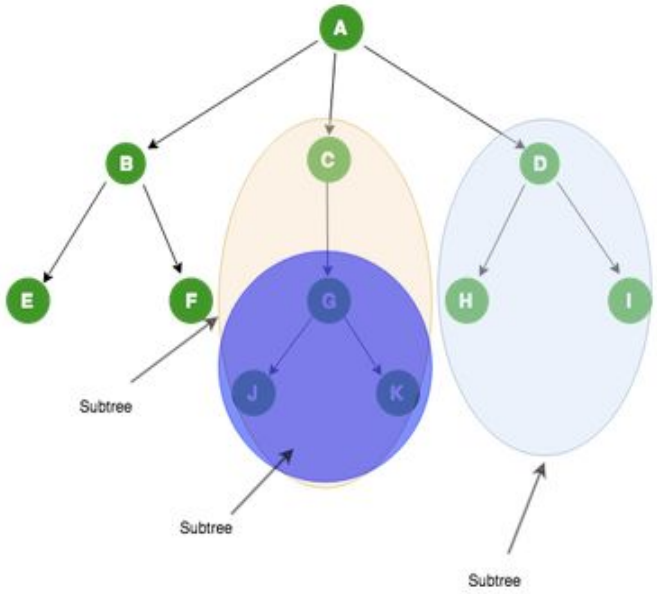


Fig 16: Illustrating Subtrees

Full Binary Trees

Full Binary Tree

A full binary tree is a tree in which each node has either 0 or 2 children. The leaf nodes have 0 children and all other nodes have exactly 2 children. Figure 2 shows an example of a full binary tree.

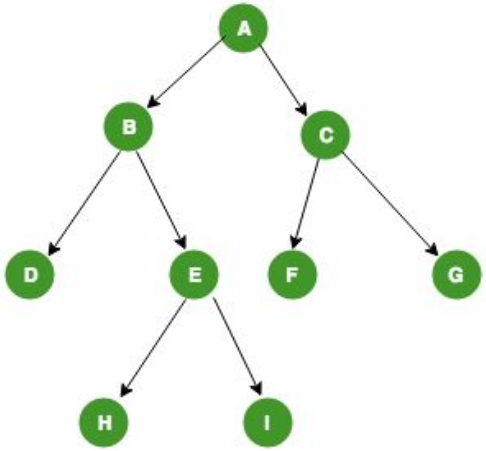


Fig 2: An example of a full binary tree

In a full binary tree, the number of leaf nodes = number of internal nodes + 1.

Complete Binary Tree

Complete Binary Tree

A complete binary tree is a tree in which every level, except possibly the last, is filled. The nodes in the last level are filled from left to right.

Properties of a complete binary tree

1. The last level of a complete binary tree
   can have 1 to 2^h nodes where h is the height of the tree.
2. The number of internal nodes is the floor of n/2, where n is the total number of nodes.

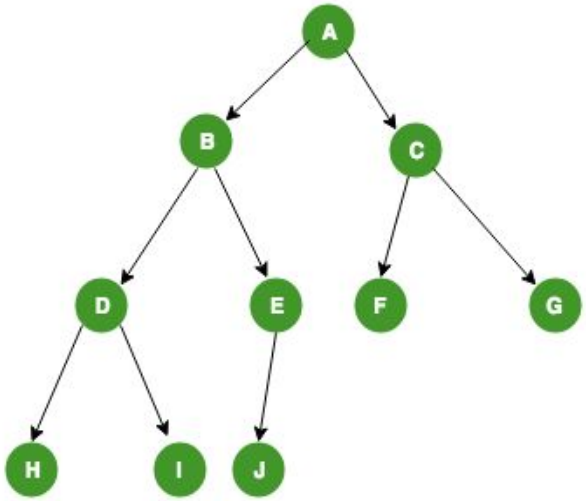


Fig 3: An example of a complete binary tree

Perfect Binary Tree

Perfect Binary Tree

A perfect binary tree is a tree where all the interior nodes have 2 children and all the leaf nodes are on the same level.

Properties of a perfect binary tree

1. The number of leaf nodes = (n + 1)/2, where n is the total number of nodes.
2. The total number of nodes = $2^{h+1} - 1$, where h is the height of the tree.

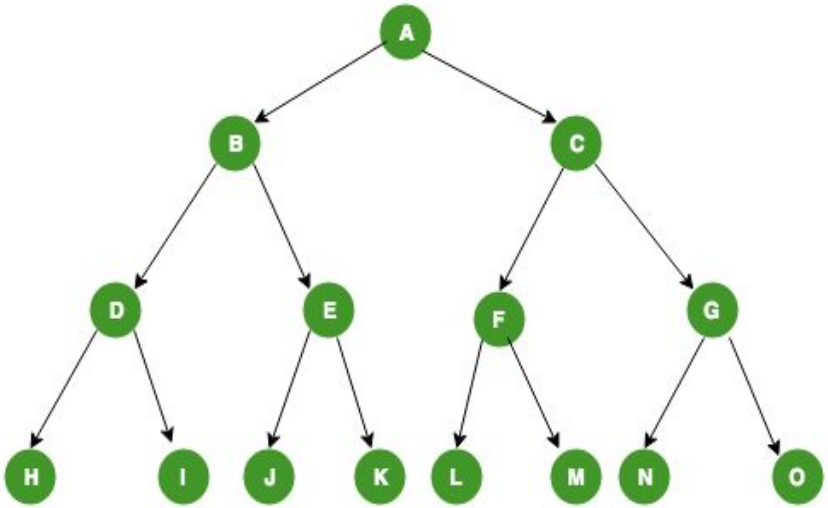


Fig 4: An example of a perfect binary tree

Balanced Binary Trees

Balanced Binary Tree

A balanced binary tree is a binary tree where the height of the left and the right subtree is differed by at most 1. This must be valid for each subtree. Examples of a balanced binary tree include AVL tree, Red-Black Tree, 2/3 Tree etc. One of the interesting property of a balanced tree is that the height of the tree is O(log n) which gives a guarantee that the insertion, deletion, and search operations are efficient.

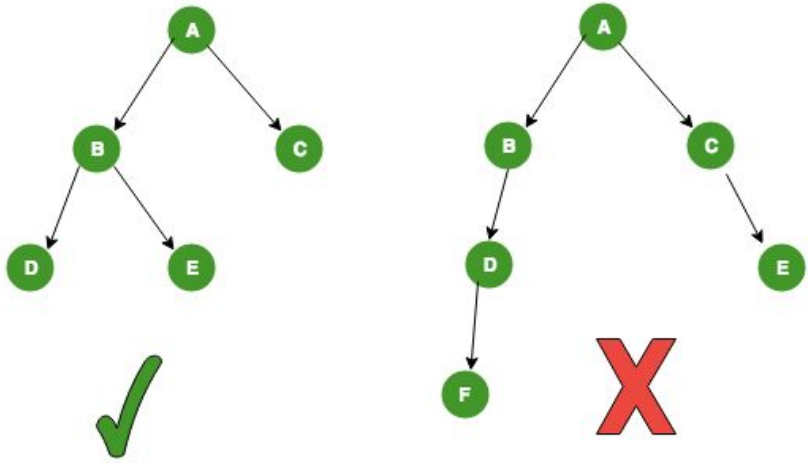


Fig 5: An example of a valid balanced binary tree (left) and invalid balanced binary tree (right)

Representation

There is two popular representation of a binary tree. One is the array representation and another is the linked list representation.

Using array

A binary tree can be implemented efficiently using an array. The array representation is most suited for a complete binary tree where the waste of the memory is minimum. We need to allocate $2^{h+1} - 1$ array items, where h is the height of the tree, that can fit any kind of binary tree. If i is the index of a parent, then the index of the left child is 2i + 1 and the index of the right child is 2i + 2. Fig 6 shows a binary tree and corresponding array representation.

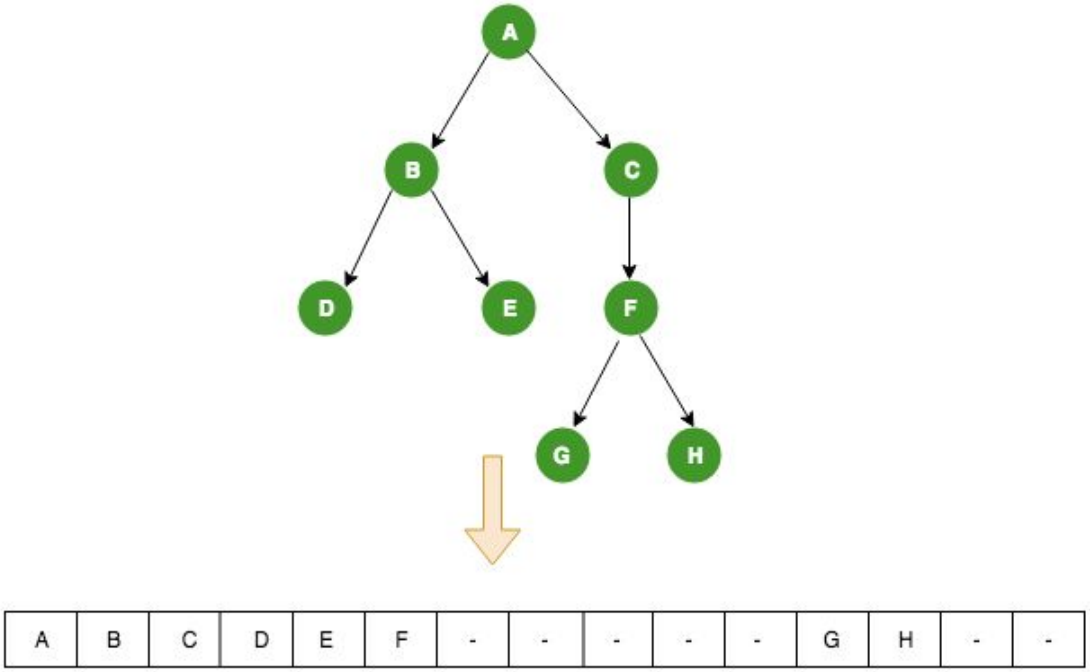


Fig 6: A binary tree and its array representation

Using linked list

In linked list representation, each node of the tree holds the reference for left and right child. Figure 7 shows a binary tree and its representation using the linked list.

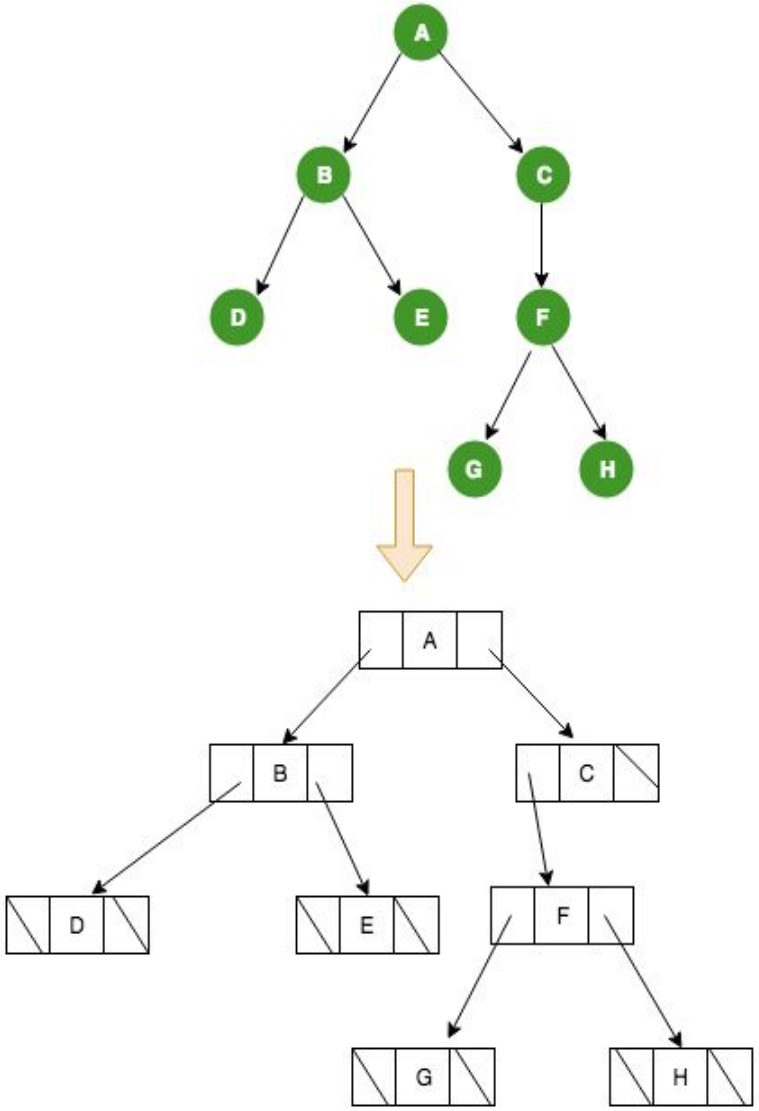


Fig 7: A binary tree implementation using linked list

Traversals

There are two types of traversal in a tree: (1) depth-first traversal and (2) breadth-first traversal.

Depth-first traversal

In depth-first traversal, we visit the child nodes before visiting the next sibling. In this way, we go deeper and deeper until there is no more child node left (i.e. we reach the leaf node). The algorithm for depth-first traversal is given below.

```
At node N,
(1) Recursively traverse its left subtree.
(2) Recursively traverse its right subtree.
(3) Visit the node N itself.
```

Depending upon the order in which these operations occur, breadth-first traversal can be further divided into three categories.

1. **Pre-Order traversal**: We visit the root node first, then recursively traverse the left subtree and recursively traverse the right subtree. The pseudocode is given below.

```
procedure pre-order(node) {
    if (node is not null) {
        print node.data;
        pre-order(node.leftnode);
        pre-order(node.rightnode);
    }
}
```

   The pre-order traversal on the binary tree given in figure 1 is : A, B, D, E, C, F ,G, H.

2. **In-Order traversal**: We visit the left subtree recursively, visit the node and finally visit the right subtree. The pseudocode is given below.

```
procedure in-order(node) {
    if (node is not null) {
        in-order(node.leftnode);
        print node.data;
        in-order(node.rightnode);
    }
}
```

   The in-order traversal on the binary tree given in figure 1 is : D, B, E, A, G, F, H, C

3. **Post-Order traversal**: We visit the left subtree recursively, visit the right subtree recursively and finally visit the node. The pseudocode is given below.

```
procedure post-order(node) {
    if (node is not null) {
        post-order(node.leftnode);
        post-order(node.rightnode);
        print node.data;
    }
```

Breadth-first traversal

It is also known as level-order traversal. In this traversal, we visit every node of a level before going to the lower level. The pseudocode for level-order traversal is given below.

```
procedure level-order(node) {
    queue.enqueue(node);

        while (queue is not empty) {
          node = queue.dequeue();
          if (node is not null) {
            print node.data;
            queue.enqueue(node.leftnode);
            queue.enqueue(node.rightnode);
          }
        }
}
```

The level-order traversal on the tree given in figure 1 is: A, B, C, D, E, F, G, H. This is illustrated in figure 8.

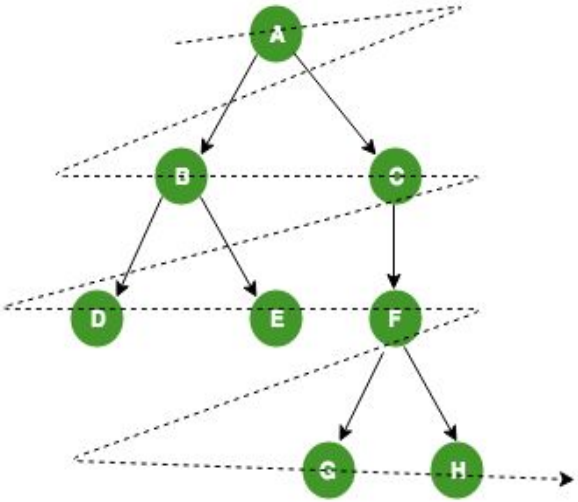


Fig 8: Illustrating level-order traversal

BST

- If you look at any node in the figure, the nodes in the left subtree are less or equal to the node and the nodes in the right subtree are greater than or equal to the node.
- The height of a randomly generated binary search tree is O(log *n*).
- Due to this, on average, operations in binary search tree take only O(log *n*) time.

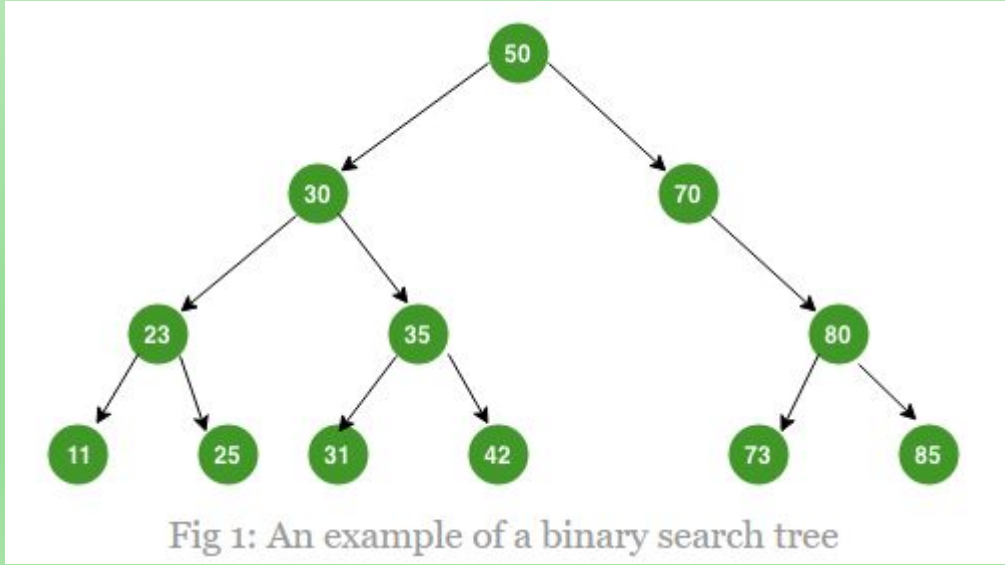


Fig 1: An example of a binary search tree

BST

- Some binary trees can have the height of one of the subtrees much larger than the other. In that case, the operations can take linear time.

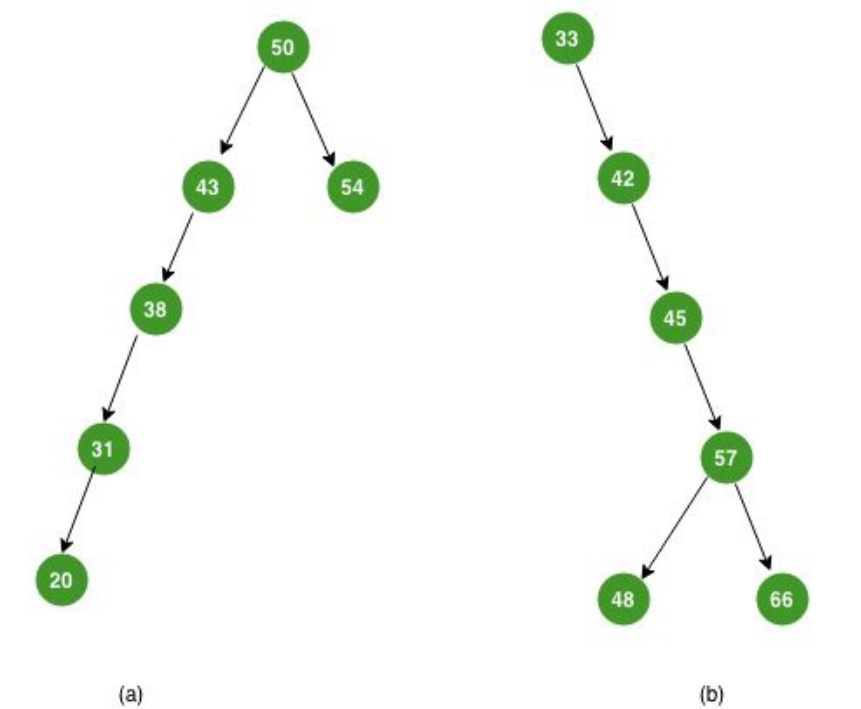


Fig 2: The examples of unbalanced binary search tree

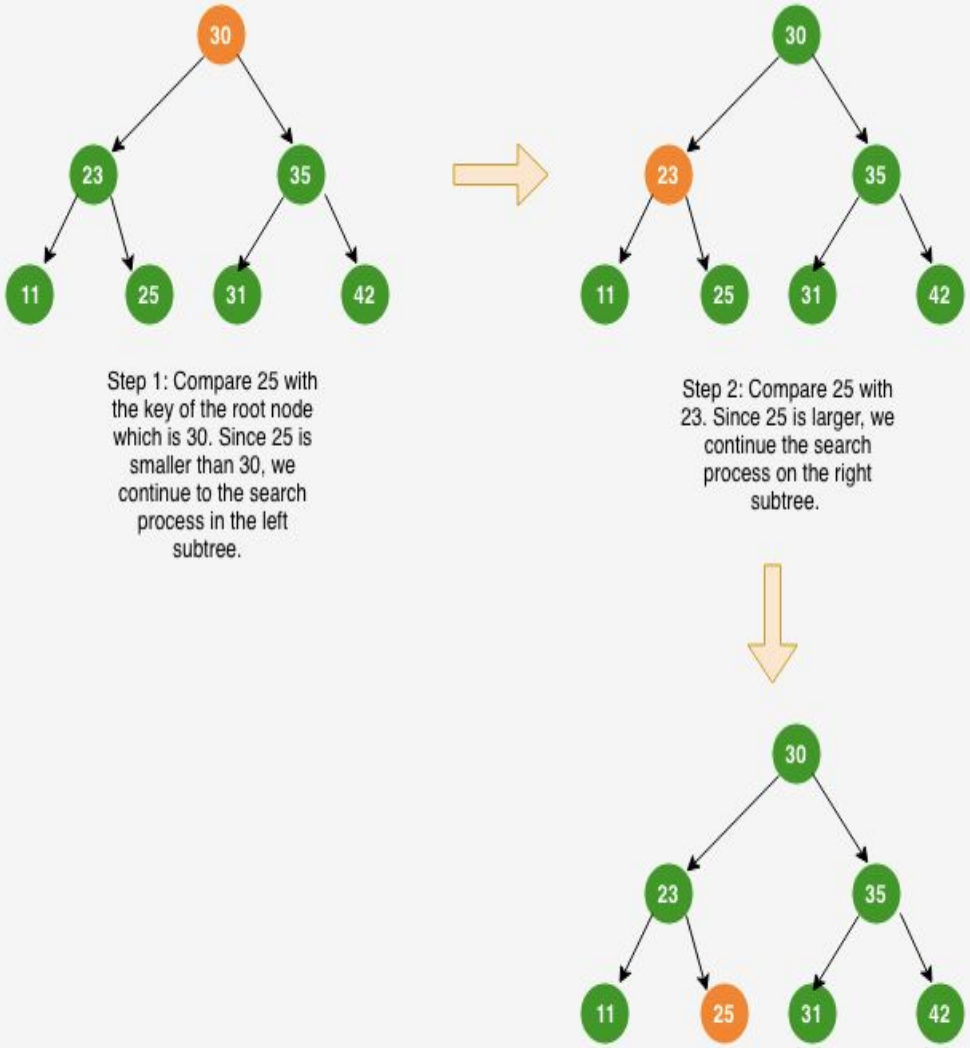


Fig 3: Illustrating the search on BST

The recursive algorithm for the search operation is given below.

```
TREE-SEARCH(x, k)
    if x == NIL or k == x.key
        return x
    if k < x.key
        return TREE-SEARCH(x.left, k)
    else return TREE-SEARCH(x.right, k)
```

Examples

Minimum and Maximum

Given a binary search tree, we can find a node whose key is minimum by following the left child pointers from the root all the way down to the leaf node. Similarly, we can find the key whose key is maximum by following the right child pointers. In other words, the left-most node of a left subtree as the minimum key and the right-most node of a right subtree has the maximum key. Figure 4 illustrates this with an example tree.

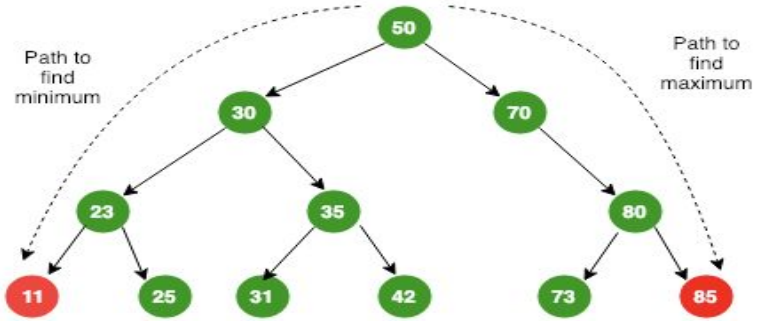


Fig 4: Illustrating the minimum and maximum operations

An algorithm to find the node with minimum key is presented below.

```
TREE-MINIMUM(x)
    while x.left != NIL
        x = x.left
    return x
```

The algorithm for the maximum is symmetric.

```
TREE-MAXIMUM(x)
    while x.right != NIL
        x = x.right
    return x
```

The running time of `TREE-MINIMUM` and `TREE-MAXIMUM` is O(h) where h is the height of the tree.

Binary Search Trees (BST)

Operations on BST

The summary of the operations I am going to discuss and their running times are outlined in the table below.

S. No.	Operation	Average Case	Worst Case
1	Search	O(log n)	O(n)
2	Minimum	O(log n)	O(n)
3	Maximum	O(log n)	O(n)
4	Predecessor	O(log n)	O(n)
5	Successor	O(log n)	O(n)
6	Insert	O(log n)	O(n)
7	Delete	O(log n)	O(n)

The algorithm for insertion operation is given below.

```
TREE-INSERT(z)
   y = NIL
   x = root
   while x != NIL
      y = x
      if z.key < x.key
         x = x.left
      else x = x.right
   z.parent = y
   if y == NIL
      root = z
   elseif z.key < y.key
      y.left = z
   else y.right = z
```

The running time of this operation is O(h) since we travel down the tree following a simple pat

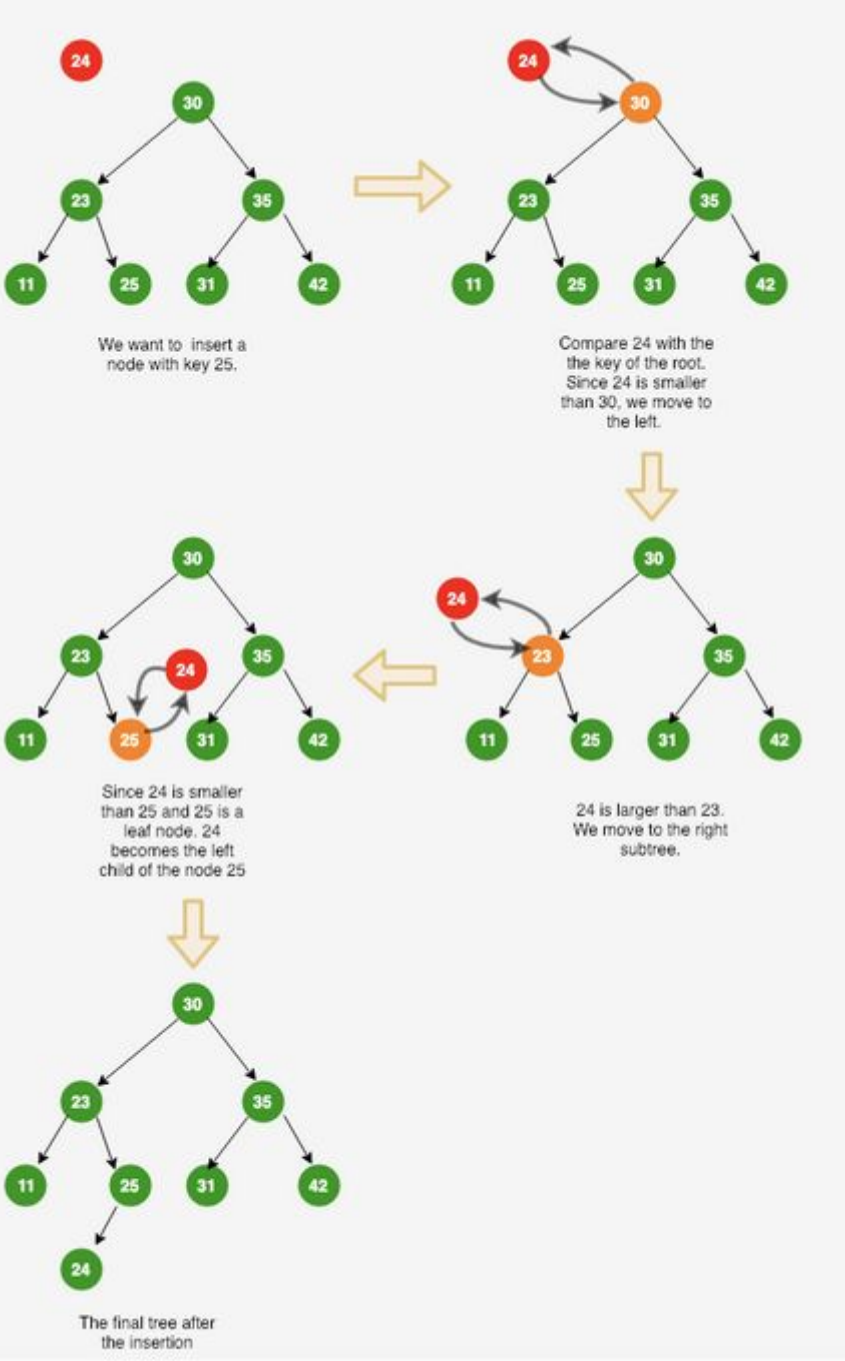


Fig 5: Illustrating the insertion operation

Deletion

The deletion operation is a little bit tricky as we need to conserve the BST property after we delete the node. We may need to transform the tree in order to conserve the BST property. We need to handle three different cases in order to delete a node x. First two cases are simple and the last one is a little bit difficult.

- **Case 1**: When x is a leaf node, we simply delete the node and change the x's parent node's left and right pointer accordingly.
- **Case 2**: When x has only one child (either left or right), we delete the node x and set the parent node's left or right pointer to the x's child node. Similarly, we adjust the parent pointer of x's child node.
- **Case 3**: When x has both the children then we do the following.
  1. We copy the minimum node y of x's right subtree and place it in the place of x.
  2. Delete y. In deleting y, we need to handle either case 1 or case 2 depending upon whether y has one child or doesn't have any children.

Let me explain the case 3 with the help of an example.

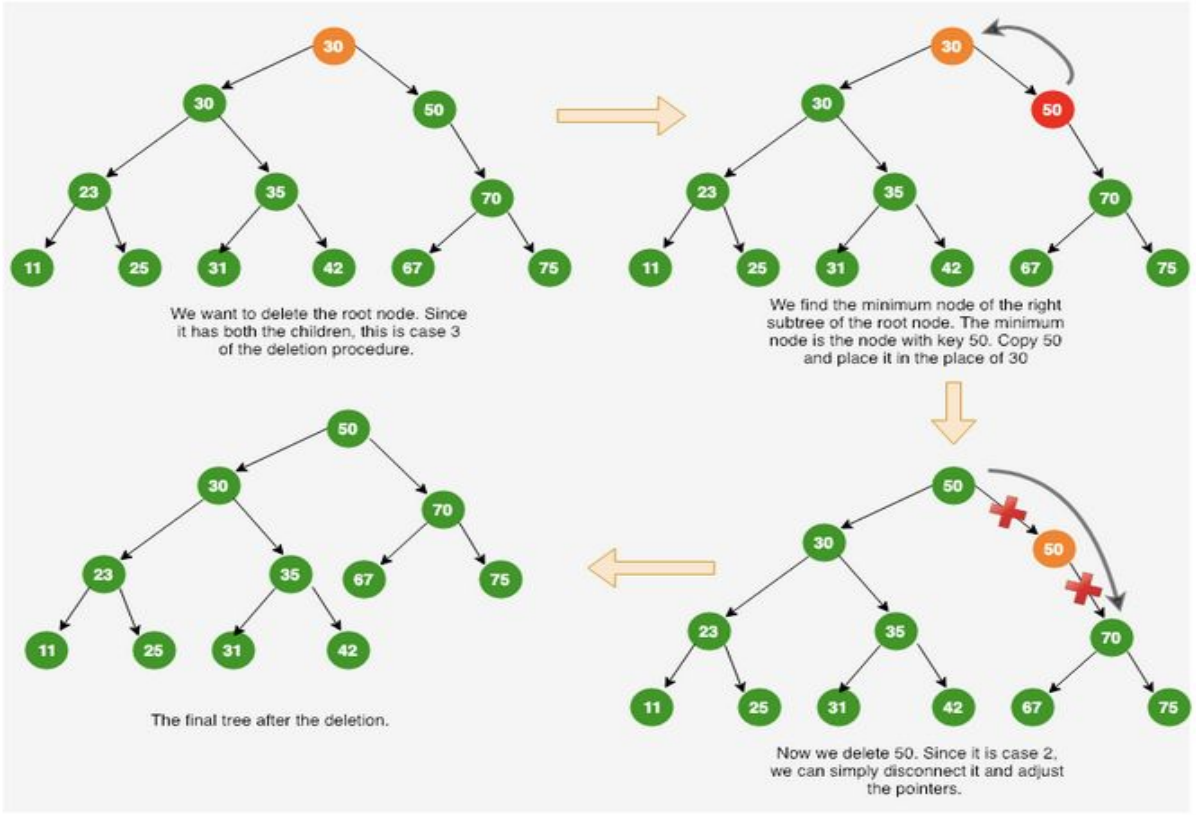


Fig 6: Illustrating the case 3 of the deletion procedure.

The algorithm for the deletion is given below.

```
TREE-DELETE(node, key)
    if node == NIL
        return node
    elseif key < node.key
        node.left = TREE-DELETE(node.left, key)
    elseif key > node.key
        node.right = TREE-DELETE(node.right, key)
    else
        //case 1
        if node.left == NIL and node.right == NIL
            delete node
            node = NIL
        // case 2
        elseif node.left == NIL
            temp = node
            node = node.right
            delete temp
        elseif node.right == NIL
            temp = node
            node = node->left
            delete temp
        // case 3
        else
            temp = TREE-MINIMUM(node.right)
            node.key = temp.key
            node.right = TREE-DELETE(node.right, temp.key)
    return node
```

The complexity of the deletion procedure is O(h).

ADT Binary Tree

Binary tree
root *left subtree* *right subtree*
createTree() *destroyBinaryTree()* *isEmpty()* *getRootData()* *setRootData()* *attachRight()* *attachLeftSubtree()* *attachRightSubtree()* *detachLeftSubtree()* *detachRightSubtree()* *getLeftSubtree()* *getRightSubtree()* *preorderTraverse()* *inorderTraverse()* *postorderTraverse()*

ADT Binary Search Tree

BinarySearchTree
root *left subtree* *right subtree*
createBinarySearchTree() *destroyBinarySearchTree()* *isEmpty()* *searchTreeInsert()* *searchTreeDelete()* *searchTreeRetrieve()* *preorderTraverse()* *inorderTraverse()* *postorderTraverse()*

References

- Queues. *Brilliant.org*. Retrieved 16:14, February 25, 2023, from https://brilliant.org/wiki/queues-basic/