8 Data Structures

Learning Outcome

Data Structures

- Illustrate the use of and implement the create, insert, and delete operations for stack and queues (linear and circular)
- Illustrate the use of and implement the create, update (edit, insert, delete) and search operations for linear linked lists
  Exclude: doubly-linked lists and circular linked lists
- Illustrate the use of and implement the create, update (edit, insert, delete) and search operations for binary trees (including binary search trees)
  Exclude: editing and deleting nodes from binary search trees
- Illustrate the use of and implement pre-order, in-order and post-order tree traversals, including the application of in-order tree traversal for binary search trees
- Illustrate the use of and implement breadth-first search and depth-first search for binary trees
- Illustrate the use of and implement hash tables (including handling collisions) with given hash functions

8.1 Overview of Collections

Collection is a group of items that we want to treat as conceptual unit. Collections can be homogeneous when all items in the collection must be of the same type, or heterogeneous when items can be of different types. For example, lists are heterogeneous in Python.

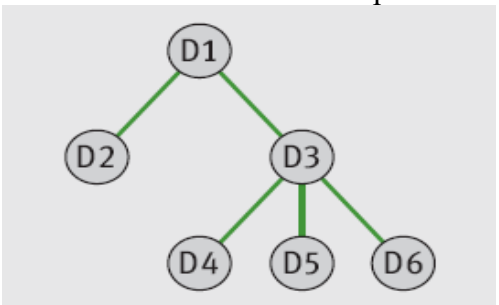
8.1.1 Linear Collections

Ordered by position

Examples: Grocery lists, Stacks of dinner plates, A line of customers waiting at a bank

8.1.2 Hierarchical Collections

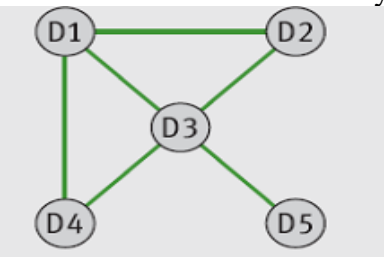
Structure reminiscent of an upside-down tree



D3's **parent** is D1; its **children** are D4, D5, and D6
Examples: a file directory system, a company's organizational tree, a book's table of contents

8.1.3 Graph Collections

Each data item can have many predecessors and many successors



D3's **neighbors** are its predecessors and successors
Examples: Maps of airline routes between cities; electrical wiring diagrams for buildings

8.1.4 Unordered Collections

Items are not in any particular order. One cannot meaningfully speak of an item's predecessor or successor

Example: Bag of marbles

8.1.5 Operations on Collections

- Traversal: This operation visits each item in a collection
- Search and retrieval: Search for a given target item or an item at a given position
- Insertion: Adds an item to a collection at a given position
- Removal: Deletes a given item or the item at a given position
- Determine the size: Determines the size of a collection – the number of items it contains

8.1.6 Abstract Data Types

- In Computer Science, collections are **abstract data types (ADTs)**
    - Benefits: Encapsulation, Reusability, Flexibility, Abstraction (like classes!)
- Examples
    - Linked List
    - Queue and Stack
    - Binary Tree

8.2 Array

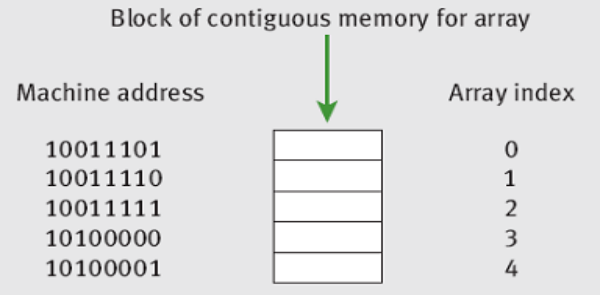
Array is a commonly used data structure. Array is the underlying data structure of a Python list, but it is more restrictive than Python lists. We have learnt and practiced enough for lists, but here let's have a recap on the concepts of array, and compare with linked structures.

8.2.1 Structure of Array

An array represents a sequence of items of the same data type (homogeneous). We can use the index to access, retrieve, store, or replace items in the array.

Random Access and Contiguous Memory

Array indexing is a random access operation: items can be accessed directly in any order with the same speed, regardless of position.



Address of an item: base address + offset. Index operation has two steps:

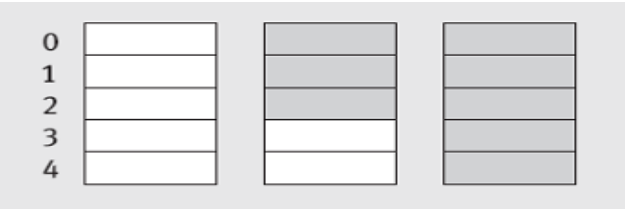
1. Fetch the base address of the array's memory block
2. Return the result of adding the (index * k) to this address, where k is the number of memory cells required by an array item.

Static Memory

Arrays are static. The capacity or length of the array is determined at compile time (before the program runs), so need to specify the size with a constant.

Physical Size and Logical Size

- The physical size of an array is its total number of array cells
- The logical size of an array is the number of items currently in it
- To avoid reading garbage, must track both sizes



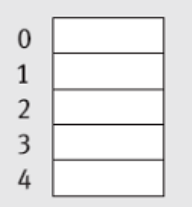
In general, the logical and physical size tell us important things about the state of the array:

- If the logical size is 0, the array is empty
- Otherwise, at any given time, the index of the last item in the array is the logical size minus 1.
- If the logical size equals the physical size, there is no more room for data in the array

8.2.2 Operations on Arrays

Indexing is the key tool in traversal, search and retrieval in an array. We now discuss the implementation of insertion and removal on arrays. In our examples, we assume the following data settings:

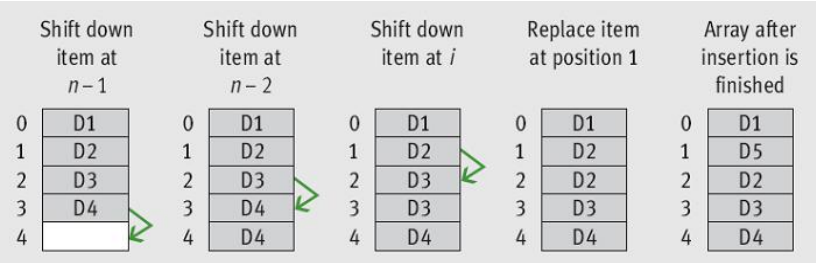
The array, A, has an initial logical size of 0 and a default physical size, or capacity, of 5.



Inserting an Item into an Array

1. Check for available space
2. Shift items from logical end of array to target index position down by one
   - To open hole for new item at target index
3. Assign new item to target index position
4. Increment logical size by one

Example: Insert item D5 at position 1 in an array of four items



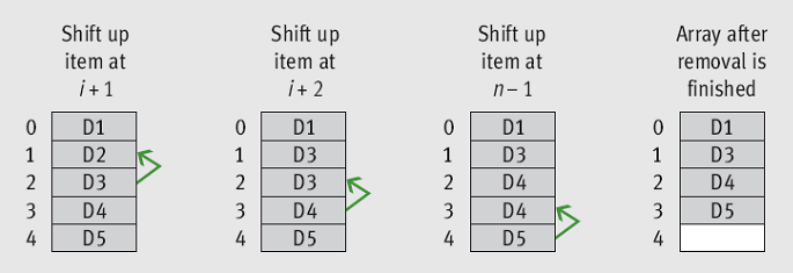
Here is the pseudo-code for the insertion operation:

```
# check for available space
IF (logicalSize = physicalSize):
      OUTPUT "No room for insertion!!"
ELSE
      # shift items down by one position
      FOR index ← logicalSize-1 TO targetIndex STEP -1
            A[index+1] ← A[index]
      ENDFOR

      # add new item and increment logical size
      A[targetIndex] ← newItem
      logicalSize ← logicalSize + 1
ENDIF
```

Removing an Item from an Array

- Shift items from next target index position to the logical end of the array up by one
  - To close hole left by removed item at target index
- Decrement logical size by one

Example: Removal of an item at position 1 in an array of five items



Here is the pseudo-code for the removal operation:

```
# shift items up by one position
FOR index ← targetIndex+1 TO logicalSize-1
     A[index-1] ← A[index]
ENDFOR

# decrement logical size
logicalSize ← logicalSize - 1
```

8.3 Linked Structure

Linked Structure, like array, is a commonly used data structure. A linked structure is a data structure that is used to implement many types of collections, including lists.

The linked list has a **head** pointer that points to the first node, and the last node points to **null**.

A linked structure allows the logical ordering of elements to be independent of their physical locations in memory, i.e. **Noncontiguous/dynamic memory** representation scheme.

8.3.1 Node

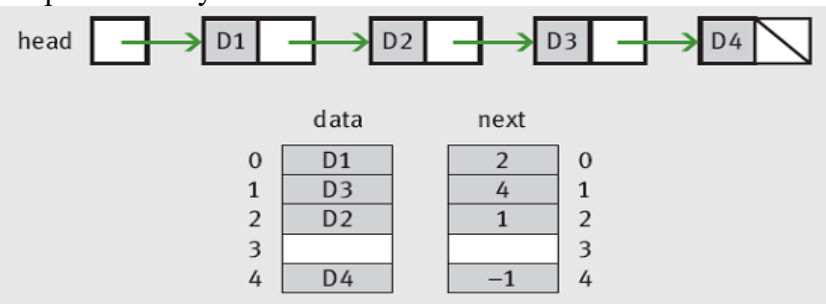
The basic unit of representation in a linked structure is a **node**. A **singly linked node** contains a data item and a link to the next node.

You can set up nodes to use noncontiguous memory in several ways:

- Using **pointers** (a **null** or **nil** represents the empty link as a pointer value): Memory allocated from the heap

- Using **references** to objects (e.g. Python)
  - In Python, **None** can mean an empty link
  - Automatic garbage collection frees programmer from managing the heap

- Using two parallel arrays



Defining a Node Class

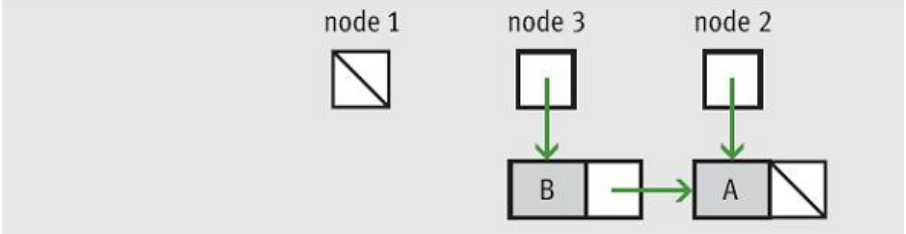
A node contains just a data item and a reference to the next node:

```
"""
File: node.py
Node class for one-way linked structures
"""

class Node:
    def __init__(self, data, next = None):
        # Instantiates a Node with default next of None
        self.data = data
        self.next = next
```

Using the Node Class

Node variables are initialized to **None** or a new **Node** object

```
# Just an empty link
node1 = None

# A node containing data and an empty link
node2 = Node("A", None)

# A node containing data and a link to node2
node3 = Node("B", node2)
```



To place the first node at the beginning of the linked structure that already contains node2 and node3:

```
node1.next = node3   raises AttributeError
```

Solution:

```
node1 = Node ("C", node3)
```

or

```
node1 = Node ("C", None)
node1.next = node3
```

To guard against exceptions:

```
if  nodeVariable != None:
     <access a field in nodeVariable>
```

Like arrays, linked structures are processed with loops.

```
from node import Node

head = None

# Add five nodes to the beginning of the linked structure
for count in range(1, 6):     # for count = 1 TO 5
    head = Node(count, head)

# Print the contents of the structure
probe = head    # initialize temporary pointer variable
while probe != None:
    print (probe.data)
    probe = probe.next
```

When the data are displayed, do they appear in the same order of their insertion?

8.3.2 Operations on Linked Structures

Indexes are an integral part of the array structure, hence almost all of the operations on arrays are index based. We shall emulate index-based operations on a linked structure by manipulating links within the structure.

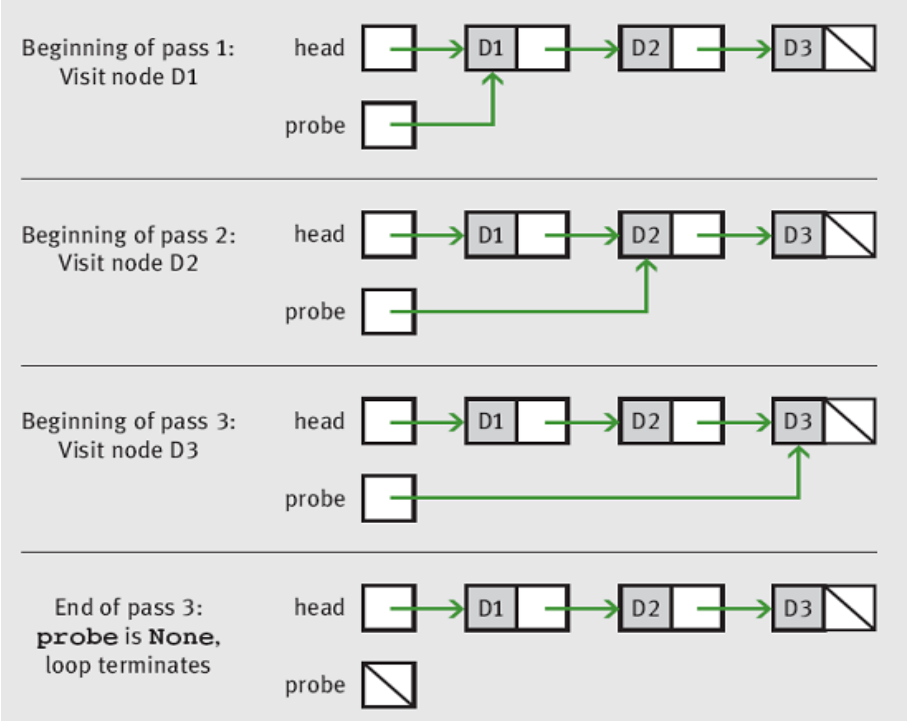
The traversal and searching operations are similar since both must traverse from the ‘start’ of the structure. The difference is whether to use index or link. The retrieval operation is straightforward for an array but more complicated for a linked structure since traversal is required. Similarly for insertion and removal of a particular item. However, no shifting of items will be required for a linked structure to perform insertion or removal.

We shall explore the implementation of these operations for various scenarios.

Traversal

In order to visit each node without deleting it, we can use a temporary pointer variable

```
probe = head
while probe != None: # not end of list
    <use or modify probe.data>
    probe = probe.next
```



Searching

Resembles a traversal, but two possible sentinels:

- Empty link - i.e. target item is not present
- Data item that equals the target item

```
probe = head
while (probe != None) and (targetItem != probe.data):
      probe = probe.next
if probe == None:
      <targetItem is not in the linked structure>
else:
      <targetItem has been found>
```

Unfortunately, accessing the i^{th} item of a linked structure is also a sequential search operation. We start at the first node and count the number of links until the i^{th} node is reached.

```
# Assumes 1 <= i <= n,
# where n is the number of nodes in the linked structure

probe = head
for count in range(i - 1):
      probe = probe.next

return probe.data
```

Since linked structures do not support random access, we can't use a binary search.

Replacement

Replacement operations employ traversal pattern

- If the target item is not present, no replacement occurs
- If the target is present, the new item replaces it

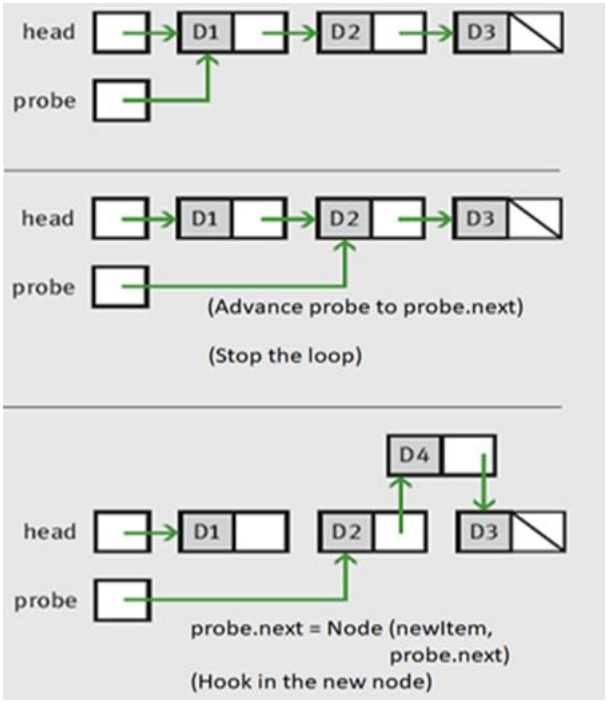
Insertion

To add a given item or the item at a given position, traverse the list to find the insert position. There are two cases to consider:

- Insert at front - updates the head pointer
- Insert at position i - updates the pointer of the $(i - 1)^{th}$ node

```
# Assumes 1 <= i <= n + 1
# where n is the number of nodes in the linked structure

if i == 1:                # case 1: insert at front
     head = Node(newItem, head)
else:                     # case 2: insert at position i
     # Search for node at position i - 1
     probe = head
     for index in range(i - 2):
          probe = probe.next
     # Insert after node at position i - 1
     probe.next = Node(newItem, probe.next)
```

The following shows a trace of the insertion of an item at position 3 in a linked structure containing three items:



Deletion

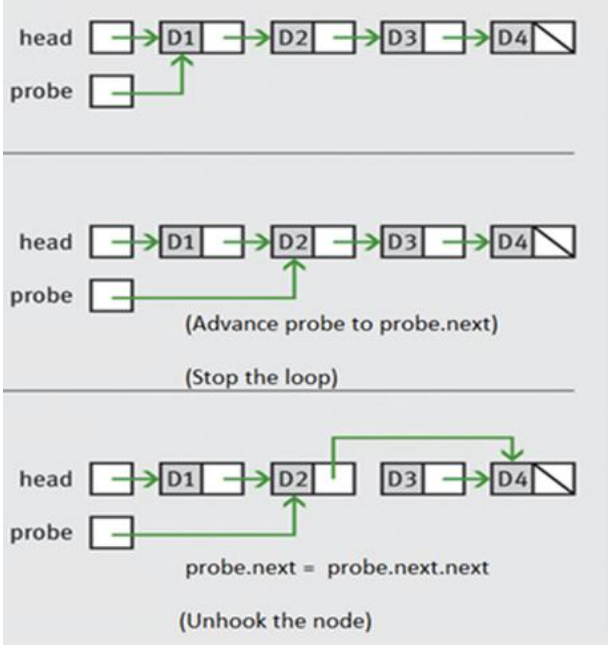
To remove a given item or the item at a given position, traverse the list to search for the node to be deleted.

As in insertion, must consider two cases:

- Delete at front - updates the head pointer
- Delete at position *i* - updates the pointer of the $(i - 1)^{th}$ node, i.e. the predecessor node

```
# Assumes that the linked structure has at least one item

if head.data == targetItem:
    head = head.next
else:
    # Search for predecessor of the node to be deleted
   probe = head
   while (probe.next != None) and (targetItem !=
 probe.next.data):
        probe = probe.next

   if probe.next == None:
        <targetItem is not in the linked structure>
   else:
        # Delete after predecessor node
        probe.next = probe.next.next
```

The following shows a trace of the removal of data item D3 in a linked structure containing four items:



Tutorial 8A

1. (a) Describe, with the aid of a diagram, the data structure called a linked list.

   (b) Describe, with the aid of diagrams, an algorithm to add a new data item into the linked list, so that this new data item occupies position **n**. You may assume that the linked list contains at least **n – 1** items before the addition.

   (c) An alternative type of list structure is one whose data items are always held in a contiguous area of store (an array). Give **one** advantage and **one** disadvantage that this has over the linked list organization.

2. A linked list Abstract Data Type (ADT) has the following operations defined:

   - create ( ) -- creates an empty linked list;
   - insert (item, p) -- insert new value, *item*, into linked list so that it is at position *p* in the linked list;
   - delete(p) -- delete the item at position *p* in the linked list;
   - read (p) -- returns the item at position *p* in the linked list;
   - length ( ) -- returns the number of items in the linked list;

The linked list is implemented by the use of a collection of nodes that have two parts: the item data and a pointer to the next item in the list. In addition, there is a *Start* pointer which points to the first item in the list.

(a) (i) Draw diagrams to show the **two** different situations that can arise when the 'delete' operation, specified above, is implemented.
(ii) Write an algorithm that could be used to implement the 'delete' operation.
(iii) Write an algorithm that could be used to implement the 'length' operation.

(b) A list, **testList**, is created and the following operations are carried out:

```
testList.insert (ben, 1)
testList.insert (jerry, 1)
testList.insert (harry, 1)
```

(i) Draw a diagram to show the state of the linked list after the above operations have been carried out.
(ii) Write the instructions that would delete the last item of **testList** after further insert operations have been carried out.

3. A program is to be written to find and print all the words in a piece of text in alphabetical order, together with the number of times each word occurs.

The data structure that has been chosen to hold this information is a linked list, with each node holding a word, the number of occurrences of that word, and a pointer to the node containing the next word in alphabetical order.

(a) Draw diagrams to illustrate the above data structure

(i) before any words in a piece of text have been read,

(ii) after reading the following text:

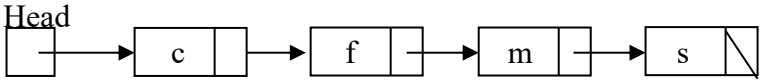
```
the man chased the cow
the cow chased the dog
the dog chased the cat
```

(b) Describe in detail an algorithm which reads a line of text, for example

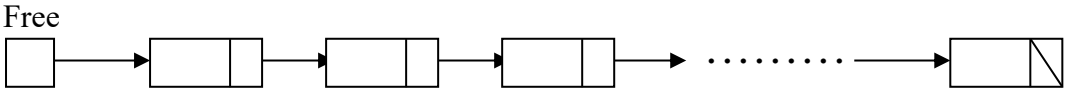
```
they all chased the rat
```

and updates the contents of the linked list. Assume an instruction is available which reads the next word from the line of text.

4. Data are to be kept in order of a key in a linked list such as that shown.



There is also a free space list.



(a) Using pseudocode, write an algorithm which will add a new element to the list. You may assume that all the keys are unique.

(b) Outline how you would delete an element from the list given its key.

5. (a) Describe, with the aid of diagrams, how names may be stored in alphabetical order in a linked list structure by using arrays. Use the names Rachel, Majid, Sian, Mary, Jonathan as example data.

(b) Show how the free space can be managed so that items can be easily added to or deleted from the linked list. Use as examples

(i) adding the name Henry to the list;
(ii) deleting the name Majid from the list.

(c) Show the contents of the arrays for **part (a)** if the linked list structure is initialized as follows:

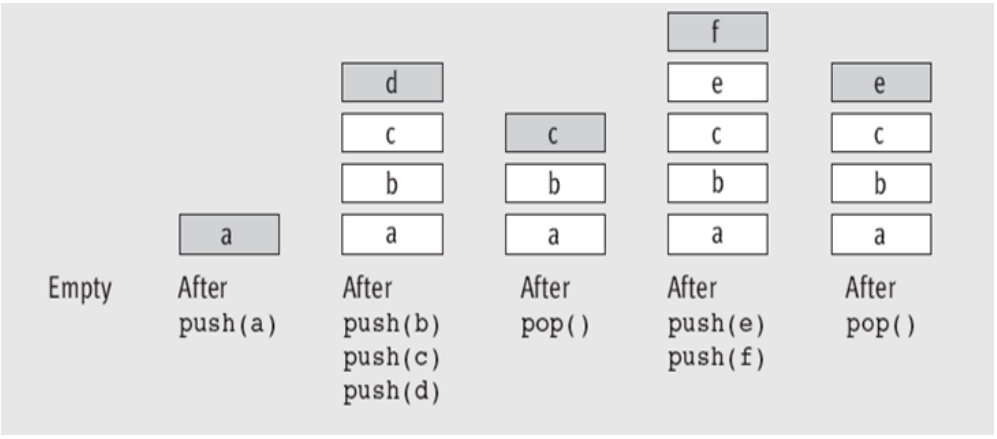
Start [-1]

Free [2]

	Name	Pointer
1		7
2		6
3		-1
4		1
5		8
6		4
7		5
8		3

8.4 Stack

LIFO (**l**ast **i**n**, f**irst **o**ut) structure in which access is completely restricted to just one end, called the **top**. It has two basic operations: **push** and **pop**.



A stack is not built into Python, though we can use a Python list to emulate a stack. For example, use list method **append** to push and **pop** to pop. However, the extra list operations violate the spirit of a stack as an ADT.

Later, we'll consider implementing stack using array (**ArrayStack**) and using linked list (**LinkedStack**).

8.4.1 Stack Interface

In addition to **push** and **pop**, a stack interface provides operations for examining the element at the top of a stack, determining the number of elements in a stack, and determining whether a stack is empty. A stack also includes an operation that return a stack's string representation. These operations are listed below, where the variable **s** refers to a stack.

STACK METHOD	WHAT IT DOES
`s.push(item)`	Inserts `item` at the top of the stack.
`s.pop()`	Removes and returns the item at the top of the stack. *Precondition*: The stack must not be empty; raises an error if that is not the case.
`s.peek()`	Returns the item at the top of the stack. *Precondition*: The stack must not be empty; raises an error if that is not the case.
`s.isEmpty()`	Returns `True` if the stack is empty, or `False` otherwise.
`s.__len__()`	Same as `len(s)`. Returns the number of items currently in the stack.
`s.__str__()`	Same as `str(s)`. Returns the string representation of the stack.

The following shows how the operations listed above affect a stack named **s**.

OPERATION	STATE OF THE STACK AFTER THE OPERATION	VALUE RETURNED	COMMENT
			Initially, the stack is empty
s.push (a)	**a**		The stack contains the single item **a**.
s.push (b)	**a b**		**b** is the top item on the stack
s.push (c)	**a b c**		**c** is the top item.
s.isEmpty ( )	**a b c**	**False**	The stack is not empty.
len (s)	**a b c**	**3**	The stack contains three items.
s.peek ( )	**a b c**	**c**	Returns the top item on the stack without removing it.
s.pop ( )	**a b**	**c**	Remove the top item from the stack and return it. **b** is now the top item.
s.pop ( )	**a**	**b**	Remove and return **b**.
s.pop( )		**a**	Remove and return **a**.
s.isEmpty ( )		**True**	The stack is empty.
s.peek ( )		**exception**	Peeking at an empty stack raises an exception.
s.pop ( )		**exception**	Popping an empty stack raises an exception.
s.push (d)	**d**		**d** is the top item.

8.4.2 Stack Application: Matching Parentheses

Compilers need to determine if the bracketing symbols in expressions are balanced correctly

EXAMPLE EXPRESSION	STATUS	REASON
`(...)...(...)`	Balanced	
`(...)...(...`	Unbalanced	Missing a closing `)` at the end.
`)...(...(...)`	Unbalanced	The closing `)` at the beginning has no matching opening `(` and one of the opening parentheses has no closing parenthesis.
`[...(...)...]`	Balanced	
`[...(...]...)`	Unbalanced	The bracketed sections are not nested properly.

Here is an approach which scans the expression and keeps checking on the matching:

- Scan expression; push left brackets onto a stack
- On encountering a closing bracket, if stack is empty or if item on top of stack is not an opening bracket of the same type, we know the brackets do not balance
- Pop an item off the top of the stack and, if it is the right type, continue scanning the expression
- When we reach the end of the expression, stack should be empty; if not, brackets do not balance

8.4.3 Stack Application: Evaluating Arithmetic Expressions

An arithmetic expression can be represented by two forms:
Infix form: each operator is located between its operands. e.g. A + B
Postfix form: an operator immediately follows its operands. e.g. A B +. This form is also called "Reverse Polish Notation".

INFIX FORM	POSTFIX FORM	VALUE
34	34	34
34 + 22	34 22 +	56
34 + 22 * 2	34 22 2 * +	78
34 * 22 + 2	34 22 * 2 +	750
(34 + 22) * 2	34 22 + 2 *	112

In both forms, operands appear in the same order. However, the operators do not.

- The infix form sometimes require parentheses; the postfix form never does.
- Infix evaluation involves rules of precedence; postfix evaluation applies operators as soon as they are encountered.
- For example:
  - Infix evaluation: **34 + 22 * 2 → 34 + 44 → 78**
  - Postfix evaluation: **34 22 2 * + → 34 44 + → 78**

To evaluate an infix expression:

- Step 1: Transform infix expressions to postfix
- Step 2: Evaluate the resulting postfix expressions

Converting Infix to Postfix

- Start with an empty postfix expression and an empty stack
  - Stack will hold operators and left parentheses
- Scan across infix expression from left to right
- On encountering an operand, append it to postfix expression
- On encountering a **(**, push it onto the stack
- On encountering an operator
  - Pop off the stack all operators with equal or higher precedence
  - Append them to postfix expression
  - Push scanned operator onto stack
- On encountering a **)**, pop operators from stack to postfix expression until meeting matching **(**, which is discarded
- On encountering the end of the infix expression, pop remaining operators from the stack to the postfix expression

Example:

INFIX EXPRESSION: **4 + 5 * 6 - 3**		**EQUIVALENT POSTFIX EXPRESSION:** **4 5 6 * + 3 -**	
PORTION OF INFIX EXPRESSION SCANNED SO FAR	**OPERATOR STACK**	**POSTFIX EXPRESSION**	**COMMENT**
			No characters have been seen yet. The stack and PE are empty.
4		**4**	Append **4** to the PE.
4 +	+	**4**	Push + onto the stack.
4 + 5	+	**4 5**	Append **5** to the PE.
4 + 5 *	+ *	**4 5**	Push * onto the stack.
4 + 5 * 6	+ *	**4 5 6**	Append **6** to the PE.
4 + 5 * 6 -	-	**4 5 6 * +**	Pop * and +, append them to the PE, and push -.
4 + 5 * 6 - 3	-	**4 5 6 * + 3**	Append **3** to the PE.
		4 5 6 * + 3 -	Pop the remaining operators off the stack and append them to the PE.

Example with parentheses:

INFIX EXPRESSION: **(4 + 5) * (6 - 3)**		**EQUIVALENT POSTFIX EXPRESSION:** **4 5 + 6 3 - ***	
PORTION OF INFIX EXPRESSION SCANNED SO FAR	**OPERATOR STACK**	**POSTFIX EXPRESSION**	**COMMENT**
			No characters have been seen yet. The stack and PE are empty.
(	**(**		Push **(** onto the stack.
(4	**(**	**4**	Append **4** to the PE.
(4 +	**(+**	**4**	Push + onto the stack.
(4 + 5	**(+**	**4 5**	Append **5** to the PE.
(4 + 5)		**4 5 +**	Pop the stack until **(** is encountered and append operators to the PE.
(4 + 5) *	*	**4 5 +**	Push * on to the stack.
(4 + 5) * (	* (	**4 5 +**	Push **(** onto the stack.
(4 + 5) * (6	* (	**4 5 + 6**	Append **6** to the PE.
(4 + 5) * (6 -	* (-	**4 5 + 6**	Push - onto the stack.
(4 + 5) * (6 - 3	* (-	**4 5 + 6 3**	Append **3** to the PE.
(4 + 5) * (6 - 3)	*	**4 5 + 6 3 -**	Pop the stack until **(** is encountered and append operators to the PE.
		4 5 + 6 3 - *	Pop the remaining operators off the stack and append them to the PE.

Evaluating Postfix Expressions

- Steps:
  - Scan across the postfix expression from left to right
  - On encountering an operator, apply it to the two preceding operands; replace all three by the result
  - Continue scanning until you reach expression's end, at which point only the expression's value remains

- To express this procedure as a computer algorithm, you use a stack of operands

- In the algorithm, **token** refers to an operand or an operator:

```
Create a new stack
While there are more tokens in the expression
    Get the next token
    If the token is an operand
        Push the operand onto the stack
    Else
        If the token is an operator
            Pop the top-two operands from the stack
            Apply the operator to the two operands just popped
            Push the resulting value onto the stack
        EndIf
    EndIf
EndWhile
Return the value at the top of the stack
```

Example:

POSTFIX EXPRESSION: 4 5 6 * + 3 –		RESULTING VALUE: 31
PORTION OF POSTFIX EXPRESSION SCANNED SO FAR	**OPERAND STACK**	**COMMENT**
		No tokens have been seen yet. The stack is empty.
4	4	Push the operand 4.
4 5	4 5	Push the operand 5.
4 5 6	4 5 6	Push the operand 6.
4 5 6 *	4 30	Replace the top-two operands by their product.
4 5 6 * +	34	Replace the top-two operands by their sum.
4 5 6 * + 3	34 3	Push the operand 3.
4 5 6 * + 3 –	31	Replace the top-two operands by their difference.
		Pop the final value.

8.4.4 Stack Application: Memory Management

- The computer's run-time system must keep track of various details that are invisible to the programmer:
  - Associating variables with data objects stored in memory so they can be located when these variables are referenced
  - Remembering the address of the instruction in which a method or function is called, so control can return to the next instruction when that function or method finishes execution
  - Allocating memory for a function's or a method's arguments and temporary variables, which exist only during the execution of that function or method

- Whenever a subroutine (function or method) is called (i.e. is activated), an **activation record** (or **stack frame**) is created to store the **current environment** for that function. Its contents include *parameters*; *local/temporary variables*; *return address* and *return value.*

- What kind of data structure should be used to store these activation records so that they can be recovered and the system reset when the function resumes execution?

  - Problem: *FunctionA* can call *FunctionB*
    FunctionB can call *FunctionC*

  - When a function calls another function, it interrupts its own execution and needs to be able to resume its execution in the same state it was in when it was interrupted.

    When *FunctionC* finishes, control should return to *FunctionB.*
    When *FunctionB* finishes, control should return to *FunctionA*.

    So, the order of returns from a function is the *reverse* of function invocations; that is, **LIFO** behavior.

  - Therefore, use a **stack** to store the activation records. Since it is manipulated at *run-time*, it is called the **run-time stack**.
- *What happens when a function is called?*

  1. **Push** a copy of its activation record **onto the run-time stack**
  2. Copy its arguments into the parameter spaces
  3. Transfer control to the starting address of the body of the function

  Thus, the **top** activation record in the run-time stack is ***always*** that of the function currently being executed.

- *What happens when a function terminates?*
  1. **Pop** the activation record of terminated function from the run-time stack
  2. Use new top activation record **to restore the environment of the interrupted function and resume** execution of the interrupted function.

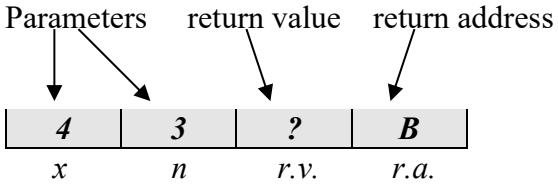
Let's look at this example of the recursive function for calculating powers.

```
def Power(x, n):
    # Power() calculates x to the nth power recursively
    if n == 0:
        return 1
    else:
        return Power(x, n-1) * x      # A


def main():
    print(Power(4, 3))                # B

main()
```

– Here we have indicated the return addresses as ***A*** and ***B***, that is, the locations of the instructions where execution is to resume when the program or function is reactivated.

– When execution of the main program is initialised, its activation record is created. This record is used to store the values of variables, actual parameters, return addresses, and so on during the time the program is active.

– When execution of the main program is interrupted by the function call ***Power (4, 3)***, the parameters ***4*** and ***3*** and the return address ***B*** (plus other items of information) are stored in the activation record, and this record is pushed onto a stack.



The function ***Power( )*** now becomes active, and an activation record is created for it.

– When the statement

return Power (x, n - 1) * x

is encountered, the execution of ***Power ( )*** is interrupted. The actual parameter ***4*** and ***2*** for this function call with parameter ***x = 4*** and ***n - 1 = 3 - 1 = 2*** and the return address ***A*** (and other items of information) are stored in the current activation record, and this record is pushed onto the stack of activation records.

Second reference to ***Power (x = 4, n = 2)*** :

4	***2***	***?***	***A***
4	***3***	***?***	***B***
x	*n*	*r.v.*	*r.a.*

Since this is a new call to ***Power ( ),*** another activation record is created, and when this function call is interrupted by the call ***Power (4, 1)***, this activation record is pushed onto the stack:

Third reference to ***Power (x = 4, n = 1)*** :

4	***1***	***?***	***A***
4	***2***	***?***	***A***
4	***3***	***?***	***B***
x	*n*	*r.v.*	*r.a.*

The call ***Power (4, 1)*** results in the creation of yet another activation record, and when its execution is interrupted, this time by the call ***Power (4, 0)***, this activation record is pushed onto the stack:

Fourth reference to ***Power (x = 4, n = 0)*** :

4	***0***	***?***	***A***
4	***1***	***?***	***A***
4	***2***	***?***	***A***
4	***3***	***?***	***B***
x	*n*	*r.v.*	*r.a.*

– Execution of ***Power ( )*** with parameters ***4*** and ***0*** terminates with no interruptions and calculates the value ***1*** for ***Power (4, 0)***. The activation record for this call is then popped from the stack, and the execution resumes at the statement specified by the return address in it:

First return from ***Power*** :

4	***0***	***1***	***A***

4	***1***	***?***	***A***
4	***2***	***?***	***A***
4	***3***	***?***	***B***
x	*n*	*r.v.*	*r.a.*

Pop stack and return to popped address ***A*** with function value ***1***

Execution of the preceding call to ***Power ( )*** with parameters ***4*** and ***1*** then resumes and terminates without interruption, so that its activation record is popped from the stack, the value ***4*** is returned, and the previous call with parameters ***4*** and ***2*** is reactivated at statement ***A***:

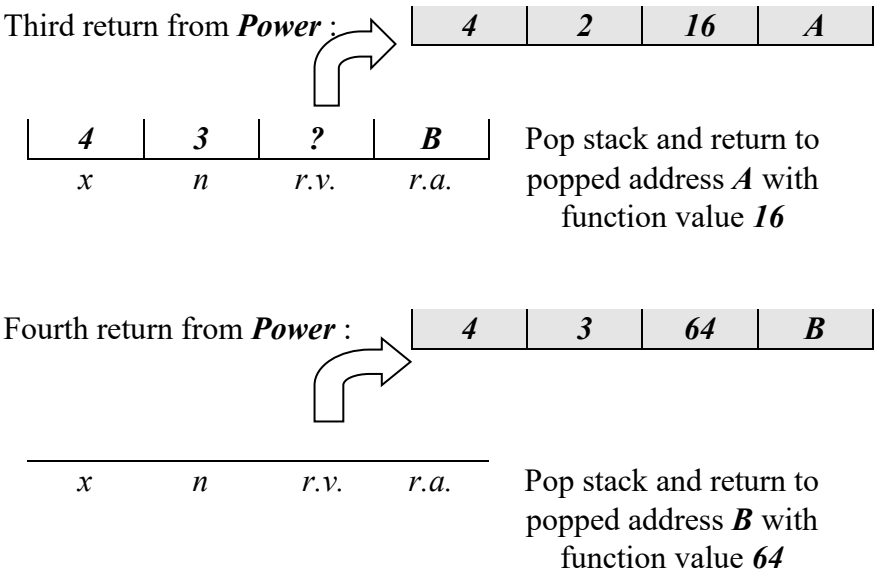
Second return from ***Power*** :

4	***1***	***4***	***A***

4	***2***	***?***	***A***
4	***3***	***?***	***B***
x	*n*	*r.v.*	*r.a.*

Pop stack and return to popped address ***A*** with function value ***4***

– This process continues until the value ***64*** is computed for the original call ***Power (4, 3),*** and execution of the main program is resumed at the statement specified by the return address ***B*** in its activation record.



8.4.5 Stack Implementation using Array

Test Driver

```
def main():
    # Test either implementation with same code
    s = ArrayStack()
    #s = LinkedStack()
    print ("Length:", len(s))
    print ("Empty:", s.isEmpty())
    print ("Push 1-10")
    for i in range(10):
        s.push(i + 1)
    print ("Peeking:", s.peek())
    print ("Items (bottom to top):", s)
    print ("Length:", len(s))
    print ("Empty:", s.isEmpty())
    print ("Push 11")
    s.push(11)
    print ("Popping items (top to bottom):", end = ' ')
    while not s.isEmpty(): print (s.pop(), end = ' ')
    print ("\nLength:", len(s))
    print ("Empty:", s.isEmpty())
```

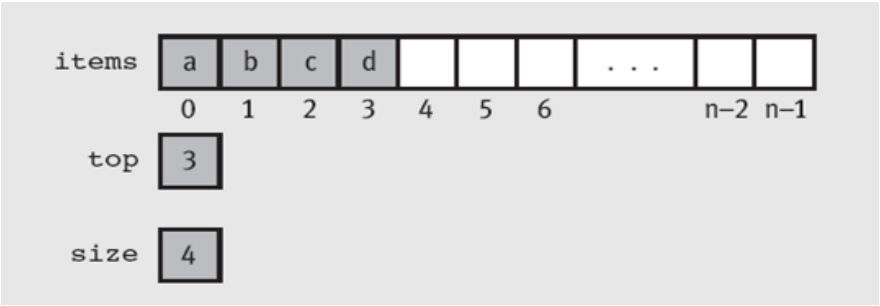
Output:

```
Length: 0
Empty: True
Push 1-10
Peeking: 10
Items (bottom to top): 1 2 3 4 5 6 7 8 9 10
Length: 10
Empty: False
Push 11
Popping items (top to bottom): 11 10 9 8 7 6 5 4 3 2 1
Length: 0
Empty: True
```

Note that the items in the stack print from bottom to top in the stack's string representation, whereas when they are popped, they print from top to bottom

Stack Implemented with Array

- Built around an array called **items** and two integers called **top** and **size**
- Initially, the array has a default capacity of **n** positions, **top** equals **-1**, and **size** equals **0**



- To **push** an item onto the stack, you increment the **top** and **size** and store the item at the location **items[top]**.
  - **size** always equals the number of items currently in the stack, whereas **top** refers to the position of the topmost item in a nonempty stack.
  - An attempt to add an item to a full stack causes an error message. (stack overflow !!)
- To **pop** the stack, you return **items[top]** and decrement **top** and **size**.
  - An attempt to delete an item from an empty stack causes an error message. (stack underflow !!)

Here is the code for **ArrayStack**:

```
class ArrayStack:
  """ Array-based stack implementation."""

  DEFAULT_CAPACITY = 12

  def __init__(self):
      self._items = [''] * ArrayStack.DEFAULT_CAPACITY
      self._top = -1
      self._size = 0

  def push (self, newItem):
      """Inserts newItem at top of stack.
      Precondition: the stack is not full."""
      if self._size == ArrayStack.DEFAULT_CAPACITY:
          print ("Stack is full. Abort operation!!")
      else:
          # newItem goes at logical end of array
          self._top += 1
          self._size += 1
          self._items[self._top] = newItem

  def pop(self):
      """Removes and returns the item at top of the stack.
      Precondition: the stack is not empty."""
      if self.isEmpty():
          print ("Stack is empty. Abort operation!!")
          return ""
      else:
          oldItem = self._items[self._top]
          self._top -= 1
          self._size -= 1
          return oldItem

  def peek(self):
      """Returns the item at top of the stack.
      Precondition: the stack is not empty."""
      if self.isEmpty():
          print ("Stack is empty. Abort operation!!")
          return ""
      else:
          return self._items[self._top]

  def __len__(self):
      """Returns the number of items in the stack."""
      return self._size

  def isEmpty(self):
      return len(self) == 0
```

```
    def __str__(self):
        """Items strung from bottom to top."""
        result = ""
        for i in range(len(self)):
            result += str(self._items[i]) + " "
        return result
```

Here is the code for application of stack on matching parentheses in section 8.4.2.
Assume that the module **stack** includes the class **ArrayStack**.

```
"""
File: brackets.py
Checks expressions for matching brackets
"""
from stack import ArrayStack

def bracketsBalance(exp):
    """exp represents the expression"""

    stk = ArrayStack()    # Create a new stack

    for ch in exp:        # Scan across the expression

         if ch in ['[', '(']: # Push an opening bracket
              stk.push(ch)


         # Process a closing bracket
         elif ch in [']', ')']:
              if stk.isEmpty( ):      # Not balanced
                   return False

              chFromStack = stk.pop( )

              # Brackets must be of same type and match up
              If (ch == ']' and chFromStack != '[') or \
                 (ch == ')' and chFromStack != '('):
                   return False

    return stk.isEmpty( )        # They all matched up

def main():
   exp = input("Enter a bracketed expression: ")
   if bracketsBalance(exp):
        print("OK")
   else:
        print("Not OK")

main()
```

8.4.6 Stack Implementation using Linked Structure

- Uses a singly linked sequence of nodes with a variable **top** pointing at the list's head, and a variable **size** to track the number of items on the stack.

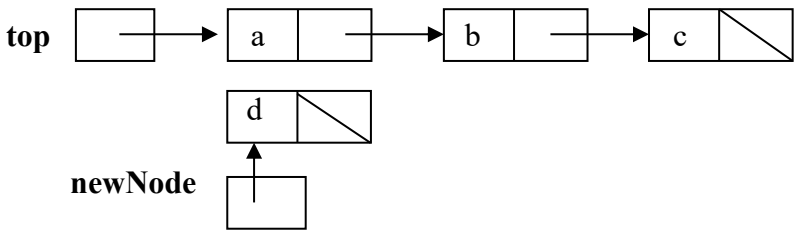
- The linked implementation requires two classes: **LinkedStack** and **Node**.

- The **Node** class contains two fields:
  - **data** : an item on the stack
  - **next** : a pointer to the next node

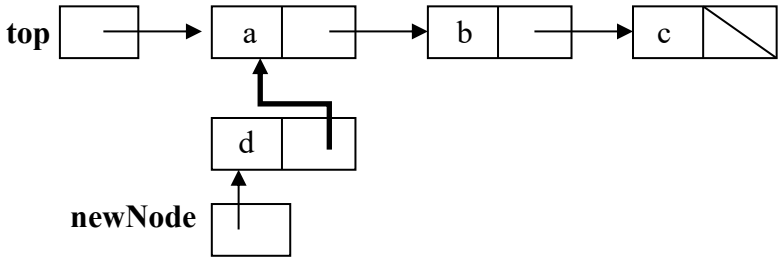
- **Pushing** and **popping** are accomplished by adding and removing nodes at the head of the list.

- **Pushing** an item onto a linked stack

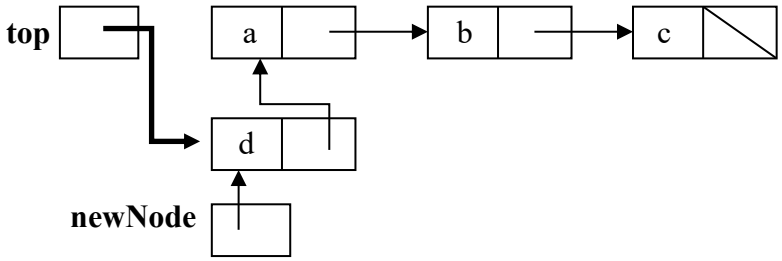
Step 1: Get a new node



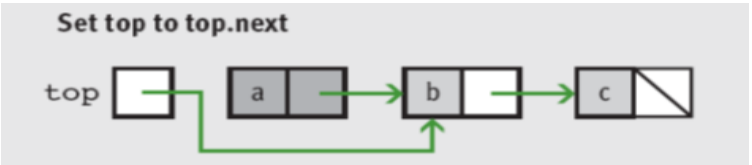
Step 2: Set **newNode.next** to **top**



Step 3: Set **top** to new node



- **Popping** an item from a linked stack



- Here is the code for **LinkedStack**:

```
from node import Node

class LinkedStack:
   """ Link-based stack implementation."""
   def __init__(self):
       self._top = None
       self._size = 0

   def push(self, newItem):
       """Inserts newItem at top of stack."""
       self._top = Node(newItem, self._top)
       self._size += 1

   def pop(self):
       """Removes and returns the item at top of the stack.
       Precondition: the stack is not empty."""
       if self.isEmpty():      # or if self._top == None:
           print ("Stack is empty. Abort operation!!")
           return ""
       else:
           oldItem = self._top.data
           self._top = self._top.next
           self._size -= 1
           return oldItem

   def peek(self):
       """Returns the item at top of the stack.
       Precondition: the stack is not empty."""
       if self.isEmpty():     # or if self._top == None:
           print ("Stack is empty. Abort operation!!")
           return ""
       else:
           return self._top.data

   def __len__(self):
       """Returns the number of items in the stack."""
       return self._size

   def isEmpty(self):
       return len(self) == 0

   def __str__(self):
       """Items strung from bottom to top."""
       result = ''
       probe = self._top
       while probe != None:
           result = str(probe.data) + ' ' + result
           probe = probe.next
       return result
```

Tutorial 8B

1. Write a program that uses a stack to test input strings to determine whether they are palindromes. A palindrome is a sequence of words that reads the same as the sequence in reverse: for example, the word ***madam*** or the sentence ***rats live on no evil star***.

2. Write a function

   def selectItem (***s***, ***n***):

   that uses stack operations to find the first occurrence of integer ***n*** on stack ***s*** and move it to the top of the stack. Maintain ordering for all other elements.

3. A stack is to be used in a high-level language program as a store for up to twelve items. The structure set up for this purpose consists of an array **STACK**, with elements **STACK[1]**, **STACK[2]** , ... , **STACK[12]** and a single integer variable **SP**. Initially **SP** is given the value zero to indicate an empty stack.

   (a) Draw a diagram to illustrate the structure after the items **ANT**, **BEE**, **COW**, **DOG**, **EEL** have been stored in the stack, in that order.

   (b) Describe carefully the algorithms which would need to be implemented in the program
   (i) to add a new item to the stack,
   (ii) to remove an item from the stack

   Your algorithms should allow for exceptional cases.

4. (a) Explain why reverse Polish form (postfix notation) might be used in preference to algebraic form (infix notation) for the representation of an algebraic expression to be processed in a computer.

   (b) Convert the following algebraic expression into reverse Polish form.

   3 * (a + b) – c * d

   (c) Using the reverse Polish expression **s 2 t + - u /**

   (i) show, with the aid of diagrams, how the computer would use a stack to calculate the results;

   (ii) write the expression in algebraic form.

5. (a) The following procedure, `TEST`, uses recursion.

```
Procedure TEST(X)
   PRINT X
   If X > 1 call TEST(X-1)
   PRINT X
END TEST
```

Procedure `TEST` is called with the parameter value of 4.

(i) What numbers will the program print, and in what order?

(ii) Show all the values of `X` held on the stack each time a `PRINT` statement is executed.

(b) If a subprogram is to be able to call itself recursively, it is usual for the values of any variables used in the subprogram to be held in a stack rather than in fixed storage. Why is this?

6. Explain what is meant by a recursive routine in a program and how a stack may be used in its implementation. To illustrate your answer, show what happens for the function call `fib(3)` where the function `fib` has a single non-negative integer parameter `n`.

```
FUNCTION fib(n)
    IF n < 2 THEN
         RETURN n
    ELSE
         RETURN fib(n-1) + fib(n-2)
    ENDIF
ENDFUNCTION
```

Show the order in which the calls to the function are made, the order in which the returns are made, and the data that are stacked at each call. Use diagrams wherever possible in your answer.

7. An Abstract Data Type (ADT) consists of both data type and associated operations.

A linked list ADT has the following operations defined:

(i) Create (x) -- creates an empty linked list *x*;
(ii) Insert (x, item, p) -- insert new value, *item*, into linked list *x* so that it is at position *p* in the linked list;
(iii) Delete (x, p) -- delete the item at position *p* in the linked list *x*;
(iv) Read (x, p) -- returns the item at position *p* in the linked list *x*;
(v) Length (x) -- returns the number of items in the linked list *x*;
(vi) IsEmptyList (x) -- returns true if linked list *x* is empty.

The linked list is implemented by the use of a collection of nodes that have two parts**:** the item data and a pointer to the next item in the list. In addition, there is a *Start* pointer which points to the first item in the list.

(a) (i) Draw diagrams to show the **two** different situations that can arise when the "Insert" operation specified above is implemented.

(ii) Write an algorithm that could be used to implement the "Insert" operation.

(b) Show how to implement the following operations for a stack ADT using the list ADT operations:

(i) create new stack;
(ii) add item on top of stack;
(iii) delete item from top of stack.

(c) A dictionary ADT is used to store a key value and a definition of that key value. Specify **three** operations for a dictionary ADT.

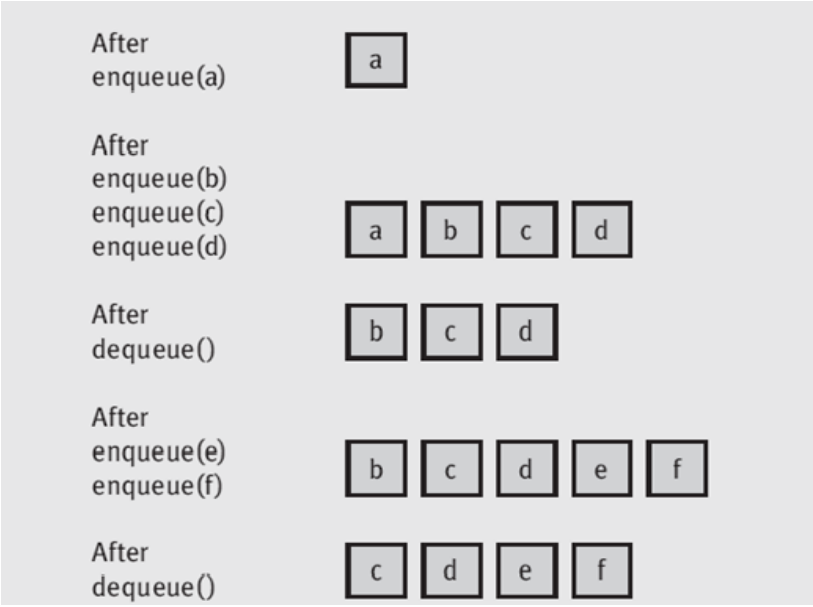
(d) State **two** advantages of using ADTs in program development.

8.5 Queue

Like stacks, queues are data structures with the following features:

- Insertions are restricted to one end, called the **rear**
- Removals are restricted to one end, called the **front**
- Queues follow a **f**irst **i**n **f**irst **o**ut (**FIFO**) structure
  - E.g. Checkout lines in stores, and airport baggage check-in lines
- Two fundamental operations:
  - **enqueue :** adds an item to the rear of a queue
  - **dequeue :** removes an item from the front

The following figure shows a queue as it might appear at various stages in its lifetime. In the figure, the queue's front is on the left, and its rear is on the right.



- Item dequeued, or served next, is always the item that has been waiting the longest
- Most queues in computer science involve scheduling access to shared resources.
  - **CPU access :** Processes are queued for access to a shared CPU
  - **Printer access :** Print jobs are queued for access to a shared laser printer.

Similar to stack, we can use a Python list to emulate a queue, use list method **append** to add an element to rear of queue and **pop** to remove an element from front of queue. However, the extra list operations violate the spirit of a queue as an ADT.

Later, we'll consider implementing queue using array (**ArrayQueue**) and using linked list (**LinkedQueue**).

8.5.1 Queue Interface

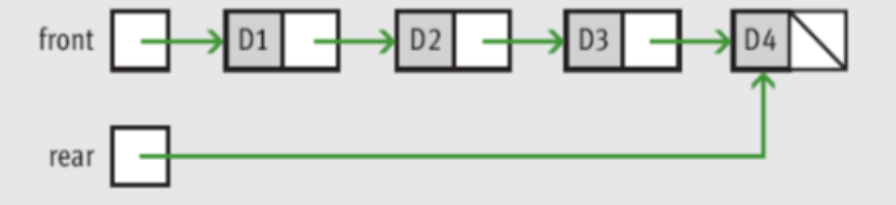
QUEUE METHOD	WHAT IT DOES
`q.enqueue(item)`	Inserts `item` at the rear of the queue.
`q.dequeue()`	Removes and returns the item at the front of the queue. *Precondition*: The queue must not be empty; an error is raised if that is not the case.
`q.peek()`	Returns the item at the front of the queue. *Precondition*: The queue must not be empty; an error is raised if that is not the case.
`q.isEmpty()`	Returns `True` if the queue is empty, or `False` otherwise.
`q.__len__()`	Same as `len(q)`. Returns the number of items currently in the queue.
`q.__str__()`	Same as `str(q)`. Returns the string representation of the queue.

The following shows how the operations listed above affect a queue named **q**.

OPERATION	STATE OF THE QUEUE AFTER THE OPERATION	VALUE RETURNED	COMMENT
			Initially, the queue is empty
q.enqueue (a)	**a**		The queue contains the single item **a**.
q.enqueue (b)	**a b**		**a** is at the front of the queue and **b** is at the rear.
q.enqueue (c)	**a b c**		**c** is added at the rear.
q.isEmpty ( )	**a b c**	**False**	The queue is not empty.
len (q)	**a b c**	**3**	The queue contains three items.
q.peek ( )	**a b c**	**a**	Returns the front item on the queue without removing it.
q.dequeue ( )	**b c**	**a**	Remove the front item from the queue and return it. **b** is now the front item.
q.dequeue ( )	**c**	**b**	Remove and return **b**.
q.dequeue ( )		**c**	Remove and return **c**.
q.isEmpty ( )		**True**	The queue is empty.
q.peek( )		**exception**	Peeking at an empty queue throws an exception.
q.dequeue ( )		**exception**	Trying to dequeue an empty queue throws an exception.
q.enqueue (d)	**d**		**d** is the front item.

8.5.2 Queue Implementation using Linked Structure

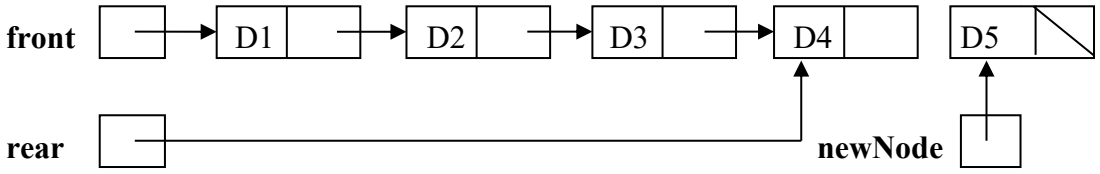
- **Enqueue** adds a node at the end
  - For fast access to both ends of a queue's linked structure, provide external pointers to both ends



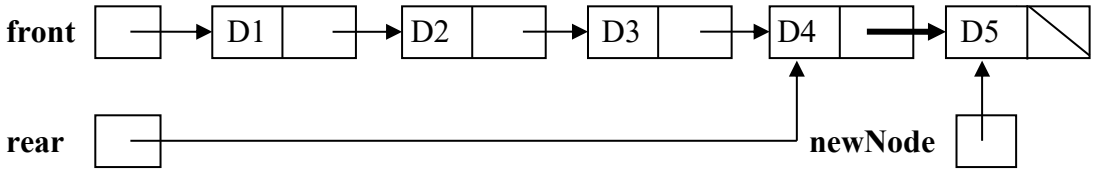
  - Instance variables **front** and **rear** of **LinkedQueue** are given an initial value of **None**
  - A variable named **size** tracks number of elements currently in queue

- During an **enqueue** operations, we create a new node, set the **next** pointer of the last node to the new node, and finally set the variable **rear** to the new node as shown below:

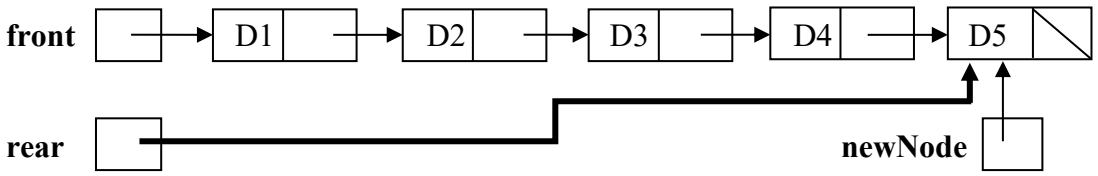
Step 1: Get a new node



Step 2: Set **rear.next** to the new node



Step 3: Set **rear** to the new node



Here is the code for the **enqueue** method:

```
def enqueue(self, newItem):
    """Adds newItem to the rear of queue."""
    newNode = Node(newItem, None)
    if self.isEmpty():
         self._front = newNode
    else:
         self._rear.next = newNode
    self._rear = newNode
    self._size += 1
```

- **Dequeue** is similar to **pop** in that it removes the first node in the sequence. However, if the queue becomes empty after a **dequeue** operation, the **front** and **rear** pointers must both be set to **None**. Here is the code for the **dequeue** method:

```
def dequeue (self):
      """Removes and returns the item at front of the queue.
      Precondition: the queue is not empty."""
      if self.isEmpty():
           print ("Queue is empty. Abort operation!!")
           return ""
      else:
           oldItem = self._front.data
           self._front = self._front.next
           if self._front  == None:
               self._rear = None
           self._size -= 1
           return oldItem
```

Here is the complete code for **LinkedQueue**:

```
from node import Node

class  LinkedQueue:
   """ Link-based queue implementation."""
   def __init__ (self):
       self._front = None
       self._rear = None
       self._size = 0

   def enqueue (self, newItem):
       """Adds newItem to the rear of queue."""
       newNode = Node(newItem, None)
       if self.isEmpty():
            self._front = newNode
       else:
            self._rear.next = newNode
       self._rear = newNode
       self._size += 1
```

```
    def dequeue (self):
        """Removes and returns the item at front of the queue.
        Precondition: the queue is not empty."""
        if self.isEmpty():
             print ("Queue is empty. Abort operation!!")
             return ""
        else:
             oldItem = self._front.data
             self._front = self._front.next
             if self._front  == None:
                 self._rear = None
             self._size -= 1
             return oldItem

     def peek (self):
        """Returns the item at front of the queue.
        Precondition: the queue is not empty."""
        if self.isEmpty():
             print ("Queue is empty. Abort operation!!")
             return ""
        else:
             return self._front.data

    def __len__ (self):
        """Returns the number of items in the queue."""
        return self._size

    def isEmpty(self):
        return len(self) == 0

    def __str__(self):
        """Items strung from front to rear."""
        result = ""
        probe = self._front
        while probe != None:
             result += str(probe.data) + " "
             probe = probe.next
        return result
```

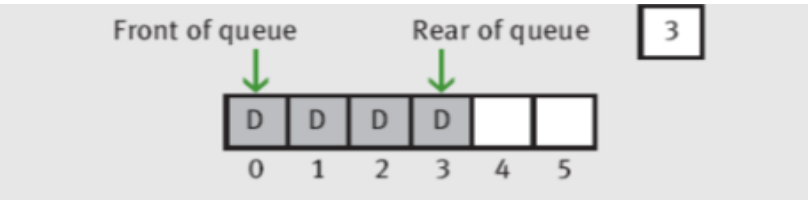
8.5.3 Queue Implementation using Array

Array implementation of a queue must access items at the logical beginning and the logical end.

- Doing this in computationally effective manner is complex
- We will approach the problem in a sequence of three attempts

A First Attempt

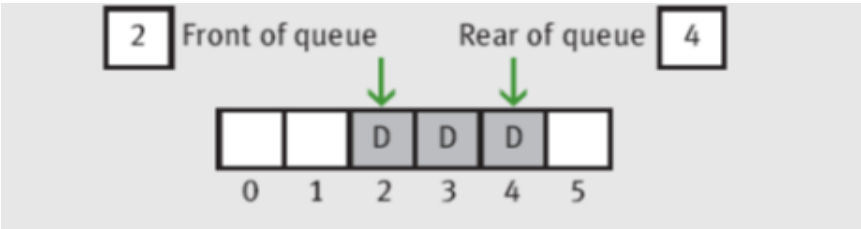
- Fixes **front** of queue at position 0
- **rear** variable points to last item at position $n - 1$, where n is the number of items in queue



- For this implementation, the **enqueue** operation is efficient. However, the **dequeue** operation entails shifting all but the first item in the array to the left.

A Second Attempt

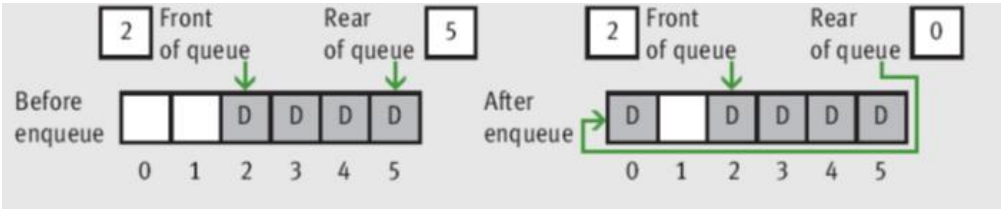
- Maintain a **front** variable that points to item at front of queue which starts at 0 and advances as items are dequeued



- For this implementation, cells to the left of the queue's front pointer are unused until we shift all elements left, which we do whenever the rear pointer reaches the end.

A Third Attempt

- Use a **circular array implementation**
  - **rear** starts at –1; **front** starts at 0
  - **front** chases **rear** pointer through the array
  - When a pointer is about to run off the end of the array, it is reset to 0
  - This has the effect of wrapping the queue around to the beginning of the array without the cost of moving any items.



- How the implementation can detect when the queue becomes full?
  - Maintain a count of the items in the queue
  - When this count equals the size of the array, the queue is full

Here is the complete code for **circular ArrayQueue**:

```
class ArrayQueue:
   """ Array-based queue implementation (circular_queue)"""

   DEFAULT_CAPACITY = 10  # Class variable applies to all queues

   def __init__(self):
       self._items = [''] * ArrayQueue.DEFAULT_CAPACITY
       self._rear = -1
       self._front = 0
       self._size = 0

   def enqueue(self, newIem):
       """Adds newItem to the rear of queue.
       Precondition: the queue is not full."""
       if self._size == ArrayQueue.DEFAULT_CAPACITY:
           print ("Queue is full. Abort operation!!")
       else:
           # end of array?
           if self._rear  == ArrayQueue.DEFAULT_CAPACITY - 1:
                self._rear = 0
           else:
                self._rear += 1

           self._items[self._rear] = newItem
           self._size += 1

   def dequeue(self):
       """Removes and returns the item at front of the queue.
       Precondition: the queue is not empty."""
       if self.isEmpty():
           print ("Queue is empty. Abort operation!!")
           return ""
       else:
           oldItem = self._items[self._front]
           # end of array?
           if self._front == ArrayQueue.DEFAULT_CAPACITY - 1:
               self._front = 0
           else:
               self._front += 1

           self._size -= 1
           return oldItem
```

```
    def peek(self):
        """Returns the item at front of the queue.
        Precondition: the queue is not empty."""
        if self.isEmpty():
            print ("Queue is empty. Abort operation!!")
            return ""
        else:
            return self._items[self._front]

    def __len__(self):
        """Returns the number of items in the queue."""
        return self._size

    def isEmpty(self):
        return len(self) == 0

    def __str__(self):
        """Items strung from front to rear."""
        result = ""
        front = self._front
        for i in range(self._size):
            result += str(self._items[front]) + " "

            if front == ArrayQueue.DEFAULT_CAPACITY - 1:
                front = 0
            else:
                front += 1

        return result
```

Tutorial 8C

1. Define a function named **stackToQueue**. This function expects a stack as an argument. The function builds and returns an instance of **LinkedQueue** that contains the elements in the stack. The function assumes that the stack has the interface described in the previous stack section. The function's postconditions are that the stack is left in the same state as it was before the function was called, and that the queue's front element is the one at the top of the stack.

2. A queue is held in an array of records with elements **q[1]** to **q[n].** The queue can contain between zero and **n** items. Write down, using pseudocode, the operations needed

   (a) When an item is added to the queue;
   (b) When an item is taken from the queue.

   Your answer should cope appropriately with the errors of adding an item to a full queue and taking an item from an empty queue.

3. One method of implementing a queue is by means of a linked list.

   (a) Draw a diagram to show how a queue can be implemented by means of a linked list.

   (b) Using this representation. Give diagrams and algorithms to show how to
      (i) add an item to the queue;
      (ii) remove an item from the queue.

8.6 Binary Tree

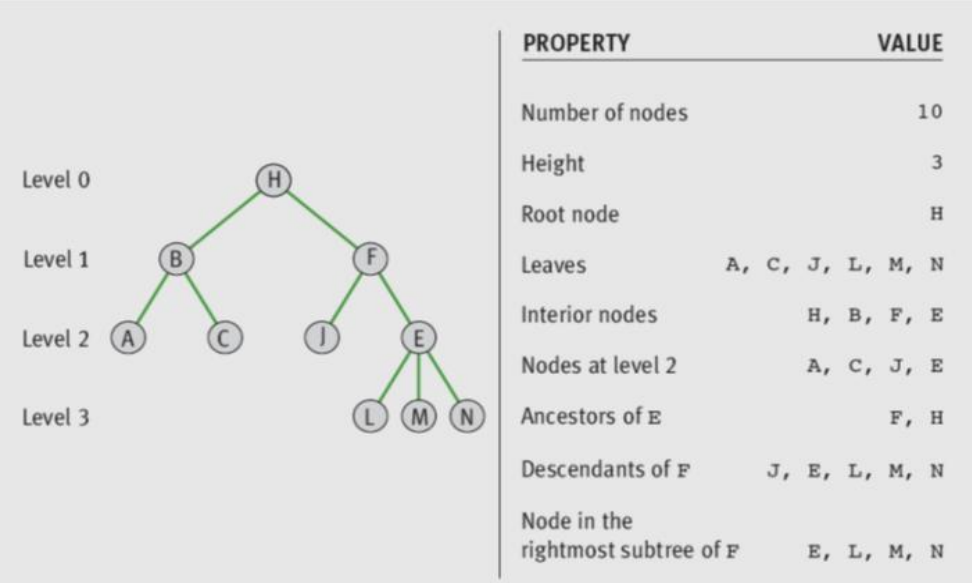
In a linear data structures (e.g. stacks, queues), all items except for the first have a distinct predecessor and all items except the last have a distinct successor. In a tree, which is a non linear data structure, the ideas of predecessor and successor are replaced with those of **parent** and **child.**

Trees have two main **characteristics**:
- Each item can have multiple children
- All items, except a privileged item called the **root**, have exactly one parent

8.6.1 Tree

TERM	DEFINITION
Node	An item stored in a tree.
Root	The topmost node in a tree. It is the only node without a parent.
Child	A node immediately below and directly connected to a given node. A node can have more than one child, and its children are viewed as organized in left-to-right order. The leftmost child is called the first child, and the rightmost is called the last child.
Parent	A node immediately above and directly connected to a given node. A node can have only one parent.
Siblings	The children of a common parent.
Leaf	A node that has no children.

TERM	DEFINITION
Interior node	A node that has at least one child.
Edge/Branch/Link	The line that connects a parent to its child.
Descendant	A node's children, its children's children, and so on, down to the leaves.
Ancestor	A node's parent, its parent's parent, and so on, up to the root.
Path	The sequence of edges that connect a node and one of its descendants.
Path length	The number of edges in a path.
Depth or level	The depth or level of a node equals the length of the path connecting it to the root. Thus, the root depth or level of the root is 0. Its children are at level 1, and so on.
Height	The length of the longest path in the tree; put differently, the maximum level number among leaves in the tree.
Subtree	The tree formed by considering a node and all its descendants.

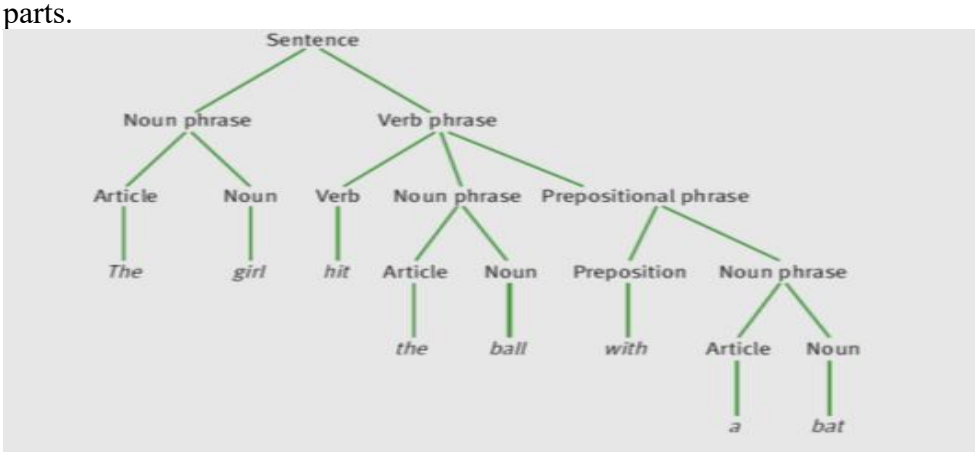


PROPERTY	VALUE
Number of nodes	10
Height	3
Root node	H
Leaves	A, C, J, L, M, N
Interior nodes	H, B, F, E
Nodes at level 2	A, C, J, E
Ancestors of E	F, H
Descendants of F	J, E, L, M, N
Node in the rightmost subtree of F	E, L, M, N

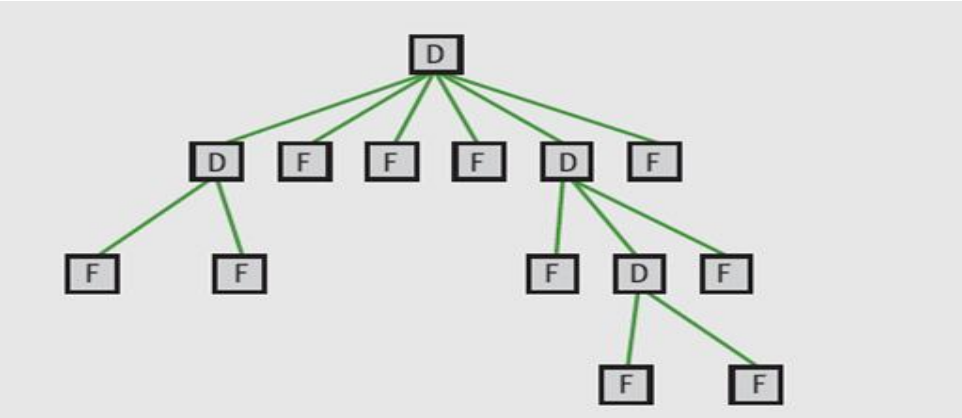
Note: The height of a tree containing one node is 0
By convention, the height of an empty tree is –1

Real-life examples:

A **parse tree** describes the syntactic structure of a particular sentence in terms of its component parts.



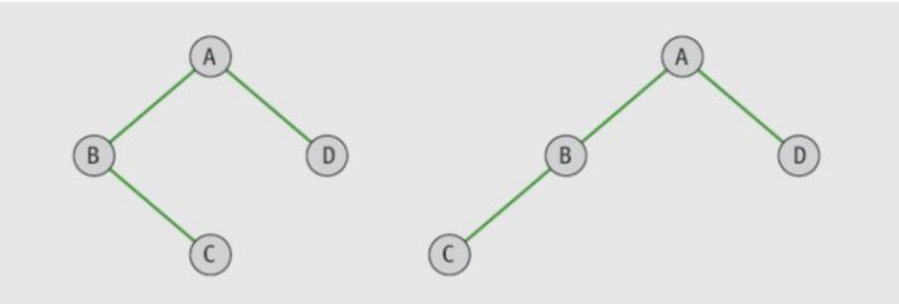
File system structures are also tree-like. The figure below shows one such structure, where the directories are labeled “D” and the files are labeled “F”



8.6.2 Binary Tree

In a **binary tree**, each node has **at most** two children:

- The **left child** and the **right child**

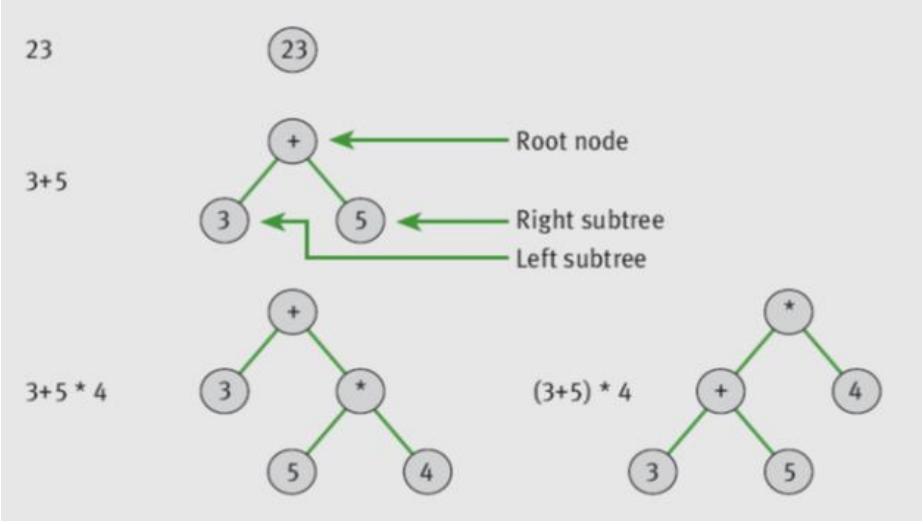


Recursive Definitions of Binary Trees

- A **binary tree** is either empty or consists of a root plus a left subtree and a right subtree, each of which are binary trees.

8.6.3 Binary Tree Application: Expression Trees

- Another way to process expressions is to build a parse tree during parsing
- An expression tree is never empty
- An interior node represents a compound expression, consisting of an operator and its operands
- Each leaf node represents a numeric operand
- Operands of higher precedence usually appear near bottom of tree, unless overridden in source expression by parentheses

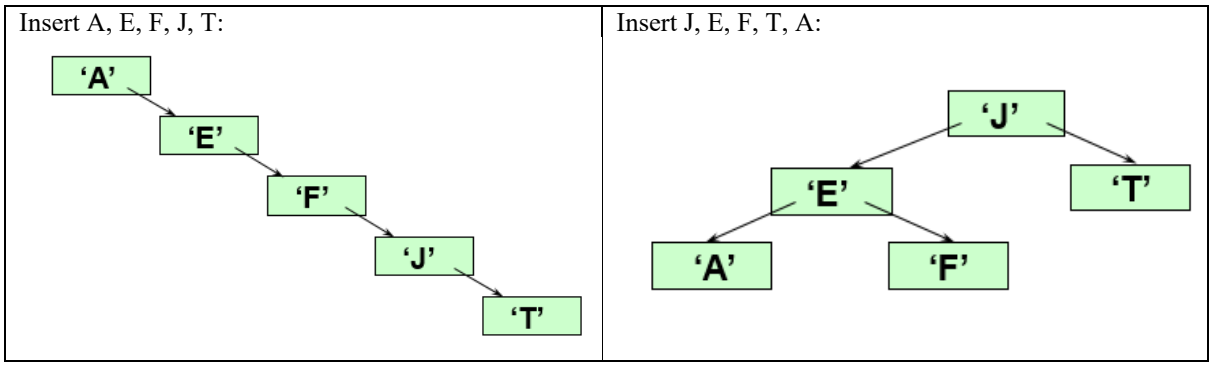


8.6.4 Binary Tree Application: Binary Search Trees

Sorted collections can also be represented as tree-like structures, called a **binary search tree**, or **BST** for short:

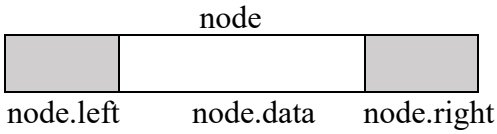
- Each node in the left subtree of a given node is less than that node, and
- Each node in the right subtree of a given node is greater than that node
- Can support logarithmic searches and insertions

The shape of a binary search tree depends on its key values and their order of insertion. For the same elements of letters A, E, F, J, T, the binary search tree depends on the order of insertion.



8.6.5 Recursive Binary Search Trees Operations

Binary search trees can be implemented using left and right pointers at each node.




```
class TreeNode:
      def __init__(self, data):
         self.left = None
         self.data = data
         self.right = None

class Tree:
     def __init__(self):
          self._root = None
```

In general, binary search tree operations can be easily implemented using recursion, which will be discussed next. However, we also need to practice on non-recursive procedures in tutorial.

Counting Number of Nodes in a Binary Search Tree

We can determine the number of nodes in the tree if we know the no. of nodes in the left subtree and the number of nodes in the right subtree.

```
# Gets the size of the tree,
# i.e. count the nodes in the tree

def CountNodes(self, tree):
      if tree == None:
            return 0
      else:
            return self.CountNodes(tree.left) +
                   self.CountNodes(tree.right)+ 1
```

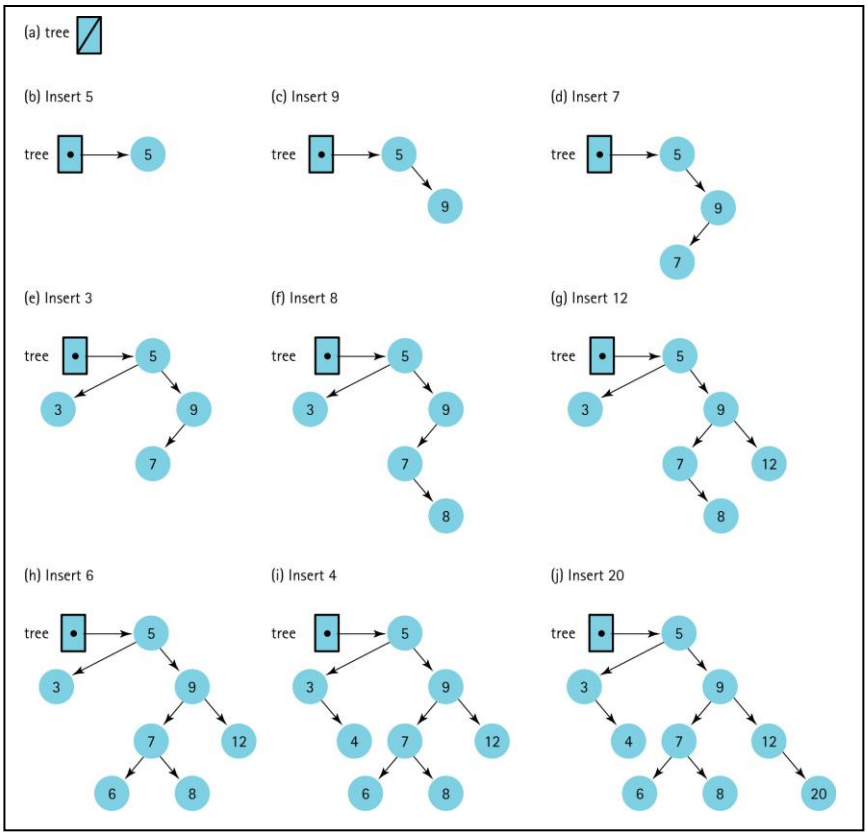
Searching a Binary Search Tree

`Search` returns `True` if the target item is in the tree; otherwise, it returns `False`

```
# Search the tree for item

def Search(self, tree, item):
    if tree == None:
        return False
    elif item < tree.data :
        return self.Search(tree.left, item)
    elif item > tree.data :
        return self.Search(tree.right, item)
    else:       # item = tree.data
        return True
```

Inserting an Item into a Binary Search Tree

- `Insert` inserts an item into the BST. Item's proper place will be in one of three positions:
  - The root node, if the tree is already empty
  - A node in the current node's left subtree, if new item is less than item in current node
  - A node in the current node's right subtree, if new item is greater than or equal to item in current node
- In all cases, an item is added as a leaf node

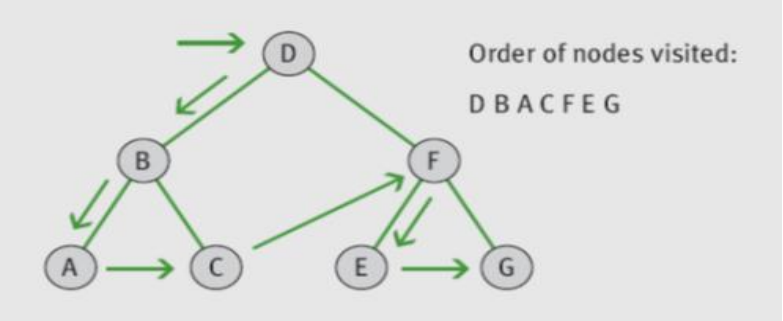



```
def Insert(self, newValue, tree):    # recursive

      if self._root == None:   # insert into empty tree
         self._root = TreeNode(newValue)

      else:
         if newValue < tree.data:
            if tree.left == None:
               tree.left = TreeNode(newValue)
            else:
               self.Insert(newValue, tree.left)

         else:                    # newValue > tree.data
            if tree.right == None:
               tree.right = TreeNode(newValue)
            else:
               self.Insert(newValue, tree.right)
```

Printing All Nodes in a Binary Search Tree

Three standard types of traversals for binary trees:

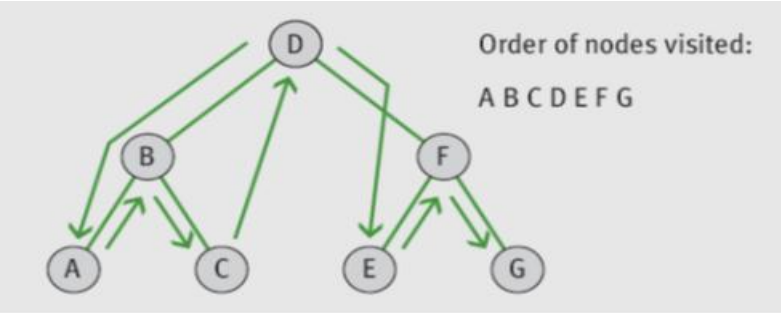
Preorder traversal: Visits root node, and then traverses left subtree and right subtree in similar way




```
# Prints the tree in Preorder

 def Preorder(self, tree):
     if tree != None:
          print(tree.data)
          self.Preorder(tree.left)
          self.Preorder(tree.right)
```

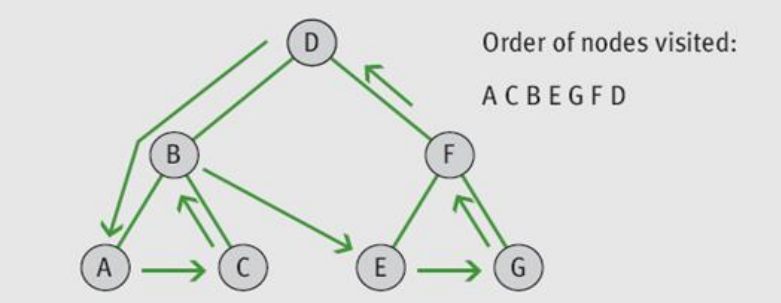
Inorder traversal: Traverses left subtree , visits root node, and traverses right subtree

- Appropriate for visiting items in a BST in sorted order




```
# Prints the tree in Inorder (ascending order)

def Inorder(self, tree):
     if tree != None:
          self.Inorder(tree.left)
          print(tree.data)
          self.Inorder(tree.right)
```

Postorder traversal: Traverses left subtree, traverses right subtree, and visits root node

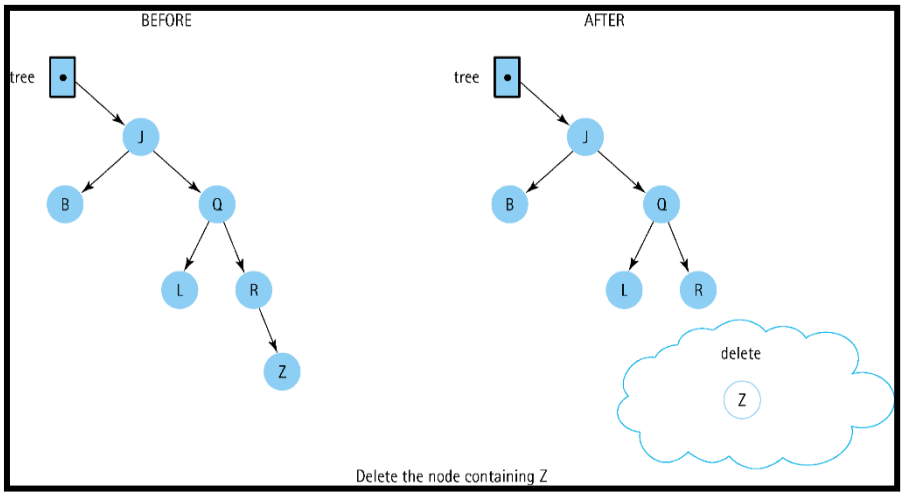



```
# Prints the tree in Postorder

 def Postorder(self, tree):
     if tree != None:
          self.Postorder(tree.left)
          self.Postorder(tree.right)
          print(tree.data)
```

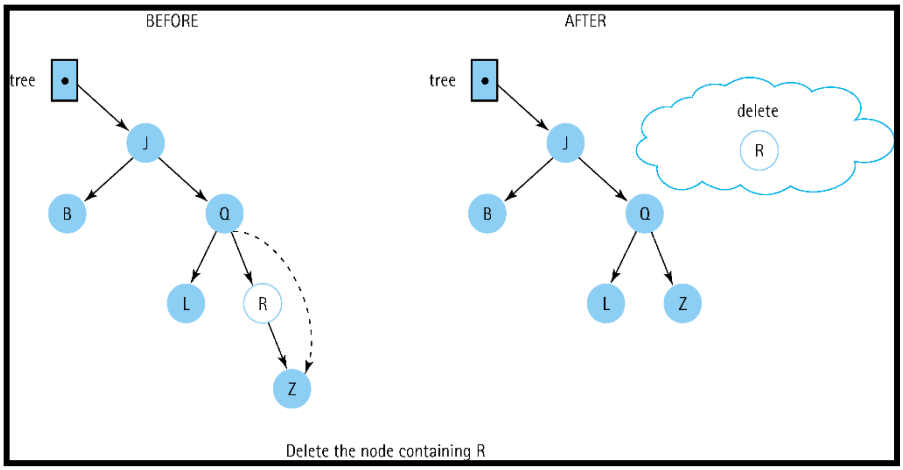
Removing an Item from a Binary Search Tree

The focus here is on the conceptual understanding of how nodes are deleted from binary search tree. Students are not required to write algorithms and programs to delete nodes from binary search tree.

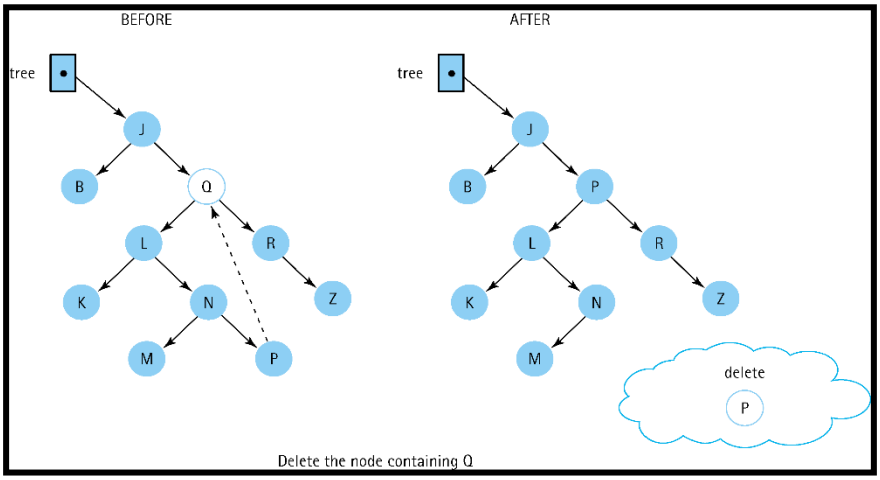
- Case 1: Deleting a Leaf Node
    - Set the parent's reference to the node to be removed to `None`



- Case 2: Deleting a Node with One Child
  - Set the parent's reference to the node to be removed to the node's only child



- Case 3: Deleting a Node with Two Children
  - Replace the data value of the node to be removed with the largest value in the left subtree and delete that value's node from the left subtree

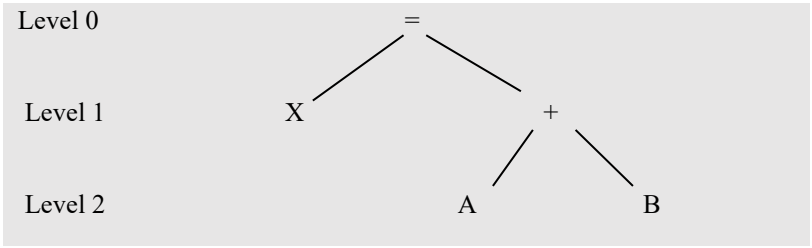


- Delete node algorithm:

```
# Delete the node referenced to by tree

if tree.left and tree.right are None  # case 1
      set tree to None
else if tree.left is None   # case 2
      set tree to tree.right
else if tree.right is None  # case 2
      set tree to tree.left
else                        # case 3
      find predecessor
      set tree.data to predecessor.data
      delete predecessor
```

8.6.6 An Array Implementation of Binary Trees

- An array-based implementation of a binary tree is difficult to define and practical only in some cases
- For binary trees, there is an elegant and efficient array-based representation
- Example:



- Elements are stored as nodes in the array
  Each node comprises a left pointer, the data and a right pointer. The pointers contain the array index of a node. -1 indicates a null pointer. The index of the root node is stored in Root.

	Index	LeftPtr	Data	RightPtr
	0	-1	X	-1
Root	1	-1	B	-1
3	2	4	+	1
	3	0	=	2
	4	-1	A	-1

- Elements are stored by level in the array

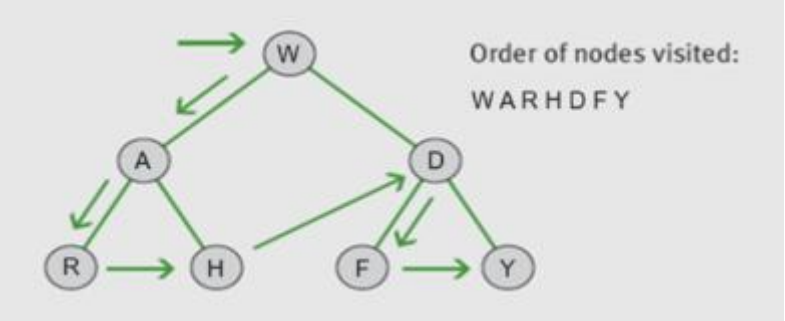
0	1	2	3	4	5	6
=	x	+	Null	Null	A	B

8.6.7 Depth-first search and Breadth-first search

Depth-first search: visits as deep as possible along one branch before backtracking to explore the next branch

Pre-order, in-order, and post-order traversal of a binary tree are a few applications of DFS.

Using the pre-order method (root, left, right):



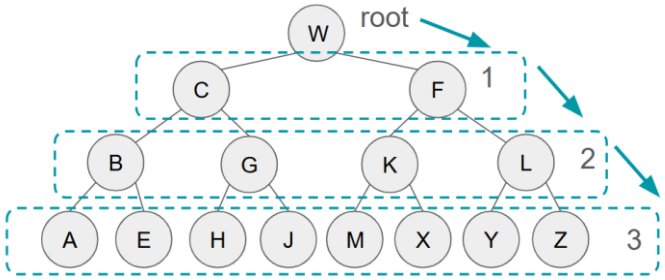
Recursive implementation of DFS:

```
# searches the tree with dfs

 def dfs(self, node, target):
     if node != None:
         if node.data == target:
             return node
         return self.dfs(node.left, target) OR
                  self.dfs(node.right, target)
```

Iterative implementation of DFS:

```
# searches the tree with dfs

 def dfs_it(self, root):
     s = Stack() # initialise a stack
     s.push(root)
     while not s.isEmpty():
         node = s.pop()
         if node.data == target:
             return node
         if node.right != None:
             s.push(node.right)
         if node.right != None:
             s.push(node.left)
     return None
```

Breadth-first search: Visits all nodes in the same level, and then visits nodes in the next level



Order of nodes visited: W C F B G K L A E H J M X Y Z (assuming L to R within the same level)

Iterative BFS with queue:

```
# traverses the tree with BFS

 def BFS(self, root, target):
     q = Queue() # initialise a queue
     q.enqueue(root)
     while not q.isEmpty():
         node = q.dequeue()
         # access and remove front of q
         if node.data == target:
             return node
         if node.left != None:
             q.enqueue(node.left)
         if node.right != None:
             q.enqueue(node.right)
     return None
```

Comparing BFS and DFS

	Breadth-first search	**Depth-first search**
Data Structure	BFS uses Queue (FIFO)	DFS uses Stack (LIFO)
Time	O(n)	O(n)
Definition	BFS is a traversal approach visiting all nodes on the same level before moving on to the next level	DFS is a traversal approach that visits as deep as possible along one branch before backtracking to explore the next branch
Order of visits	Level by level	Branch by branch
Better for	Searching nodes closer to the root	Searching nodes further from the root
Ideal Target	Target node is near the root	Target node is further from the root
Max Memory	Widest width of a level	Longest path (depth of binary tree)
Applications	Finding the shortest path in unweighted graphs	Exploring the entire tree with low memory (preorder, inorder, postorder)

Tutorial 8D

1. Write an iterative algorithm for searching an item in a binary search tree.

```
def SearchItem (tree, item) :
# Function:        Searches item in tree
# Postcondition:   If found, returns true; otherwise, returns false
```

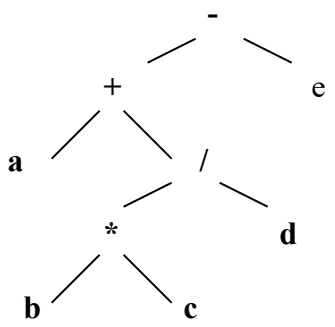
2. Write an iterative algorithm for inserting an item in a binary search tree.

```
def InsertItem (tree, item) :
# Function:        Adds item to tree
# Precondition:    item is not in the tree
# Postconditions:  item is in tree
                   Binary search property is maintained
```

3. For each sequence of characters, draw the binary search tree and traverse the tree using inorder, preorder, and postorder.
   (a) M, T, V, F, U, N
   (b) F, L, O, R, I, D, A

4. An arithmetic expression involving the binary operators add (+), subtract (-), multiply (*), and divide (/) can be represented using a **binary expression tree**.

   In a binary expression tree, each operator has two children that are either operands or sub-expressions. Leaf nodes contain an operand and non-leaf nodes contain a binary operator. The left and right subtrees of an operator describe a sub-expression that is evaluated and used as one of the operands for the operator. For instance, the expression **a + b * c / d - e** corresponds to the binary expression tree.



   (a) Perform preorder, inorder, and postorder traversals of the binary expression tree. What relationship exists among these scans and the prefix, infix, and postfix (RPN) notation for the expression?

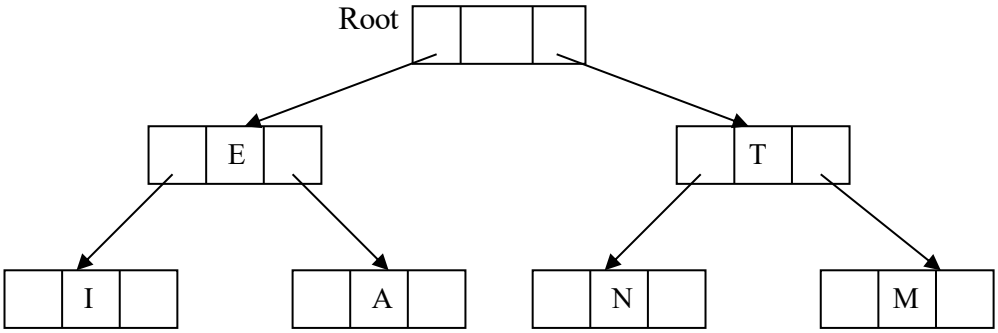
   (b) For each arithmetic expression, draw the corresponding expression tree. By scanning the tree, give the prefix, infix, and postfix form of the expression.

   (i) a - b * c + d (ii) a + b - c * d + e

5. In Morse code each letter of the alphabet is assigned a unique combination of dots and dashes. For example, the letters A, B, C and D are coded as follows.

A	• –
B	– • • •
C	– • – •
D	– • •

This coding system can be represented in a binary tree as follows. Each node, except for the root node, contains a letter of the alphabet. The position of each letter in the tree is determined by its Morse code. Moving from one node to another down the tree is done by traversing either a left branch or a right branch. If a left branch corresponds to a • and a right branch corresponds to a –, the first three levels of the tree look like this.



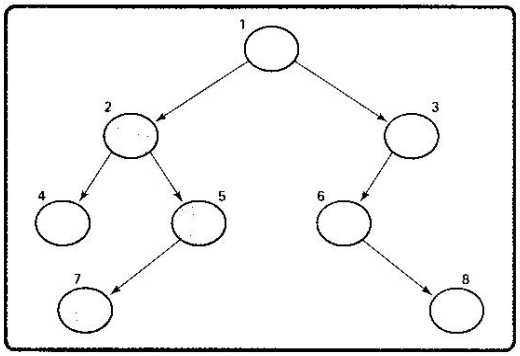
(a) What are the Morse codes for the letters I and N? Explain how your answers are derived from the tree.

(b) Draw a diagram of the binary tree which shows clearly the position of the letters D, C and B in the tree.

(c) Assuming that the complete binary tree for Morse codes exists, describe in detail an algorithm which uses the tree to read the Morse code for a letter and to print the letter.

(d) Explain why this binary tree representation is not the most suitable data structure for performing English to Morse code conversion. Describe a better alternative, and explain how the Morse code of a letter could be found.

6. Examine the following binary search tree and answer the questions. The numbers on the nodes are *labels*; they are not data values within the nodes.



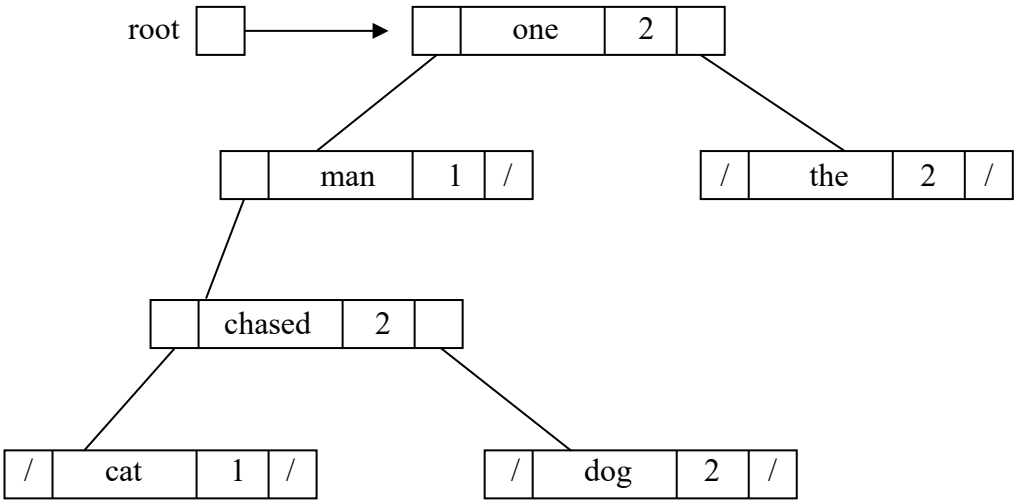
(a) If an item is to be inserted whose data value is less than the data value in node 1 but greater than the data value in node 5, where will it be inserted?

(b) If node 1 is to be deleted, the data value in which node could be used to replace it?

(c) 4 2 7 5 1 6 8 3 is a traversal of the tree in which order?

(d) 1 2 4 5 7 3 6 8 is a traversal of the tree in which order?

7. All the words in a piece of text are to be stored in a binary tree. Each node will store a word and the number of times that word has occurred in the text so far. As each new word is read, it is added to the tree. After processing the first ten words of a particular piece of text, the tree looks like this



(a) (i) Explain why the first five words of the text could not have been
the man chased the dog

(ii) Show the contents of the tree after the next five words
they never chased the rat
have been processed.

(b) The following recursive algorithm **printtree** has been suggested as a way of writing all the words in the text in alphabetical order.

```
if tree is not empty then
    Write out the word stored in the current node
    Write out the right sub-tree using printtree
    Write out the left sub-tree using printtree
Ifend
```

Unfortunately, **printtree** does not write out the words in the required order.

(i) Write down the order in which the words from the above tree would be printed by **printtree**.

(ii) Explain how to rewrite **printtree** so that the words are printed in alphabetical order.

(c) If, after building a complete tree, all the words had to be printed in order of decreasing frequency, describe briefly how this could be accomplished efficiently.

8.7 Hash Table

A hash table is a data structure where the index of each item is determined by a hash function of the item itself. When searching a hash table, we can simply search at the determined location of the item.

8.7.1 Hash Function

As an illustration, suppose that up to 25 key-value pairs made up of an integer and a string — integers in the range 0 – 999, and strings of fruit names — are to be stored in a hash table. This hash table can be implemented as an array.

The function h defined by h(i) that determines the index of the pair with integer i in the hash table is called a **hash function**.

For example,

$$h(i) = i \text{ modulo } 25$$

or in Python,

```
# division by 25 method
def h(i):
    return (i % 25)
```

because this function always produces an integer in the range 0 through 24. This is the **hash value.** The key-value pair [52, “Banana”] thus is stored at index 2, since h(52) = 52 % 25 = 2. Similarly, [129, “Carrot”], [500, “Durian”], [273, “Eggplant”], and [49, “Apple”] are stored at indexes 4, 0, 23, and 24 respectively.

	Hash Table
table[0]	[500, “Durian”]
table[1]	None
table[2]	[52, “Banana”]
table[3]	None
table[4]	[129, “Carrot”]
table[5]	None
.	.
.	.
.	.
.	.
.	.
.	.
table[23]	[273, “Eggplant”]
table[24]	[49, “Apple”]

8.7.2 Collision Strategies

The above hash table has a big problem: **collisions**.

For example, to store [77, "Grape"], it should be stored at h(77) = 77 % 25 = 2, but this location is occupied by [52, "Banana"]. Other keys like 2, 27 and 102, all integers 25k + 2, where k is any positive integer, will have the same hash value of 2. Thus, to resolve such collisions, we can use **linear probing** or **chaining**.

Linear Probing (or Linear Probe Open Addressing)

In this method, a linear search of the table begins at the index where a collision occurs and continues until an empty slot is found in which the item can be stored.

Thus, in the preceding example, when [77, "Grape"] collides with the value [52, "Banana"] at index 2, we move to index 3 to store [77, "Grape"]. For [102, "Honeydew"], we follow the **probe sequence** consisting of indexes 2, 3, 4, and 5 to find the first available empty element and thus store [102, "Honeydew"] at index 5.

If the search reaches the bottom of the table at index 24, we continue searching for an empty spot from index 0. For example, [123, "Jackfruit"] will collide with [273, "Eggplant"] at index 23. The probe sequence 23, 24, 0, 1 locates the first empty slot at index 1. Thus [123, "Jackfruit"] is stored at index 1.

	Hash Table
table[0]	[500, "Durian"]
table[1]	[123, "Jackfruit"]
table[2]	[52, "Banana"]
table[3]	[77, "Grape"]
table[4]	[129, "Carrot"]
table[5]	[102, "Honeydew"]
.	.
.	.
.	.
.	.
.	.
.	.
table[23]	[273, "Eggplant"]
table[24]	[49, "Apple"]

Let's initialise a hash table!

```
# initialise empty array of size 25 with None
table = [None] * 25
```

```
# function to insert with linear probing
def insert(table, key, value):
    hash_value = h(key)

    if table[hash_value] == None:
        table[hash_value] = [key, value]
        # insertion successful
        return hash_value
    else:
        # linear probing
        ptr = hash_value + 1
        while ptr != hash_value:
            if table[ptr] == None:
                table[hash_value] = [key, value]
                # insertion successful
                return hash_value
            ptr += 1
            if ptr == len(table):
                ptr = 0
        # no empty slots found, insertion unsuccessful
        return -1
```

Hash Table Search with Linear Probing
To determine if a specified key-value pair is in this hash table, we first apply the hash function to the integer key to compute the index at which this value should be found. There are three cases to consider.

1. Element empty: value not in table
2. Element contains matching value: search successful!
3. Element contains another value: start linear probing.

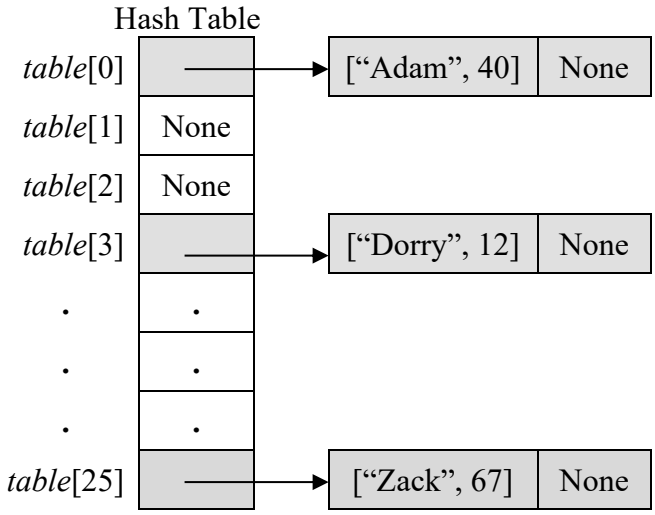
The third case happens due to the way collisions are handled. In this case, we begin a "circular" linear search from this index and continue until either the item is found, or we reach an empty element or the starting index, which indicates that the item is not in the table.

For linear probing, whenever collisions occur, the colliding values are stored in indexes that should be reserved for items that hash directly to these indexes. This approach makes subsequent collisions more likely, thus compounding the problem.

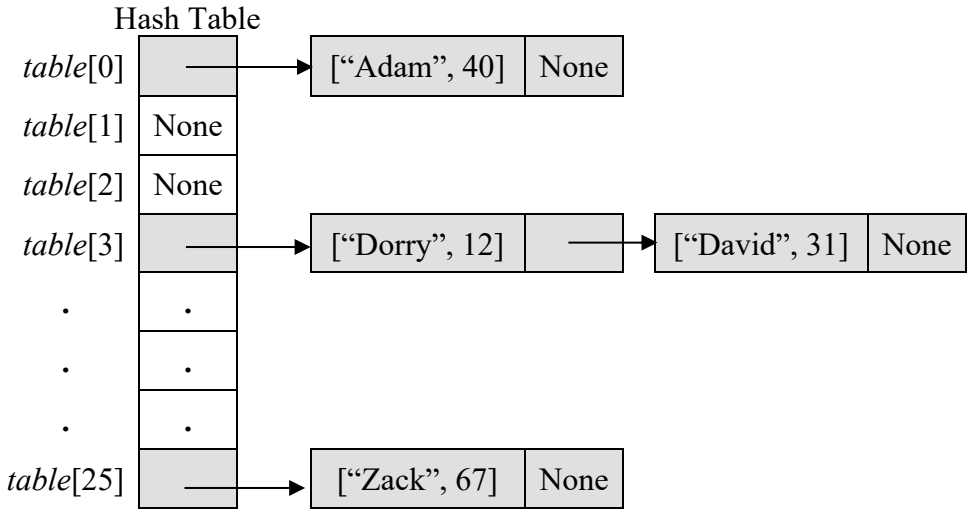
Chaining (or Chaining with Separate Lists)

A better approach, known as **chaining,** uses a hash table that is an array of linked lists that store the items. To illustrate, suppose we wish to store a collection of names and ages, we can use their names as the keys and ages as the values. This can be implemented with an array of 26 linked lists, initially empty, and the simple hash function h(name) = ord(name[0]) – ord('A') ; that is, h(name) is 0 if name[0] is 'A', 1 if name[0] is 'B', . . . , 25 if name[0] is 'Z'.

Thus, for example, [“Adam”, 40] and [“Dorry”, 12] are stored at indexes 0 and 3 respectively.



When a collision occurs, we simply insert the new item into the appropriate linked list. For example, since h(“David”) = h(“Dorry”) = ord(‘D’) – ord(‘A’) = 3, a collision occurs when we attempt to store [“David”, 31], and thus we add a new node with this name to the linked list at index 3 :



Hash Table Search with Chaining

To search a chained hash table, apply the hash function to the target item, then search the linked list at that index.

Chaining is generally faster than linear probing since only items sharing the same index (hash value) are searched. Unlike linear probing's fixed-length table, chaining uses dynamically allocated linked lists limited only by available memory. Its main drawback is the extra memory needed for node pointers. But overall, chaining is preferred as it has faster search.

Choosing a Hash Function

The hash function directly affects collision frequency. A poor function causes clustering. For example, hashing by first letter means names starting with 'T' form much longer lists than those starting with 'Z', increasing search time for T-names than for Z-names.

A better approach distributes keys more uniformly, such as averaging the first and last letters:

h(name) = (ord(first letter) + ord(last letter)) // 2 – ord('A')

A good hash function should:

1. distribute keys **uniformly** to minimise collisions and ensure search operations are fast and efficient
2. be **deterministic**. Same key should always generate the same hash value with the hash function, allowing the hash table to store and retrieve the key value pairs consistently
3. be **fast** to compute. Hashing algorithm should generate hash values efficiently, allowing for fast insertion, deletion and search operations on the hash table

Tutorial 8E

1. Using a hash table with eleven locations and the hashing function h(i) = i % 11, show the hash table that results when the following integers are inserted in the order given: 26, 42, 5, 44, 92, 59, 40, 36, 12, 60, 80 . Assume that collisions are resolved using

   (a) linear probing.

   (b) chaining.

2. Customers are identified by ID numbers. These ID numbers and names are to be stored in a hash table. The hashing function to be used is

   ```
   Address ← IDnumber MOD Max
   ```

   The hash table is implemented as a two-dimensional array with `Max` elements.

   Task 1
   Write program code to:

   - Read ID numbers and names from a text file and store them in a hash table. For the purpose of testing the program, `Max` is to be set to the value 20.
     Assume different IDs will hash to different addresses (no collisions).
   - Print out the contents of the hash table in the order in which the elements are stored in the array.

   Use `KEYS1.TXT` to test your program code.

   Task 2
   Amend your program code so that collisions can be managed using open hashing. This means a collision is resolved by searching sequentially from the hashed address for an empty location and storing the ID and name at this empty location.

   Use `KEYS2.TXT` to test your program code.

   Task 3
   Add code to your Task 2 program.
   The program is to:

   - Take as input an ID number
   - Search the hash table and output the address (index number) of the hash table where the ID was found and the corresponding name.

   Use `KEYS2.TXT` to test your program code.
   Run the program three times. Use the following inputs: 37, 77 and 97.