Teaching London Computing

A Level Computer Science

Topic 9: Data Structures

William Marsh
School of Electronic Engineering and Computer Science
Queen Mary University of London

Aims

- Where do lists and dictionaries come from?
- Understand the problem
- Introduce the following data structures
  - Linked list
  - Binary search tree
  - Hash sets
  - Graphs

Syllabus – OCR

	are used to represent text.
1.4.2 Data Structures	a) Arrays (of up to 3 dimensions), records, lists, tuples. b) The following structures to store data: linked-list, graph (directed and undirected), stack, queue, tree, binary search tree, hash table. c) How to create, traverse, add data to and remove data from the data structures mentioned above. *(This can be **either** using arrays and procedural programming **or** an object-oriented approach).*

2.3 Algorithms	
The use of algorithms to describe problems and standard algorithms	
2.3.1 Algorithms	a) Analysis and design of algorithms for a given situation. e) Algorithms for the main data structures, (Stacks, queues, trees, linked lists, depth-first (post-order) and breadth-first traversal of trees). f) Standard algorithms (Bubble sort, insertion sort, merge sort, quick sort, Dijkstra's shortest path algorithm, A* algorithm, binary search and linear search).

Syllabus – AQA

4.2.1.4 Abstract data types/data structures

Content	Additional information
Be familiar with the concept and uses of a: • queue • stack • list • graph • tree • hash table • dictionary • vector.	Be able to use these abstract data types and their equivalent data structures in simple contexts. Students should also be familiar with methods for representing them when a programming language does not support these structures as built-in types.
Be able to distinguish between static and dynamic structures and compare their uses, as well as explaining the advantages and disadvantages of each.	
Describe the creation and maintenance of data within: • queues (linear, circular, priority) • stacks • hash tables.	

Syllabus – AQA

4.2.4 Graphs

4.2.4.1 Graphs

Content	Additional information
Be aware of a graph as a data structure used to represent more complex relationships.	
Be familiar with typical uses for graphs.	
Be able to explain the terms: • graph • weighted graph • vertex/node • edge/arc • undirected graph • directed graph.	
Know how an adjacency matrix and an adjacency list may be used to represent a graph.	
Be able to compare the use of adjacency matrices and adjacency lists.	

Data Structures?

"I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."

Linus Torvalds, 2006

Problem: Arrays → Lists

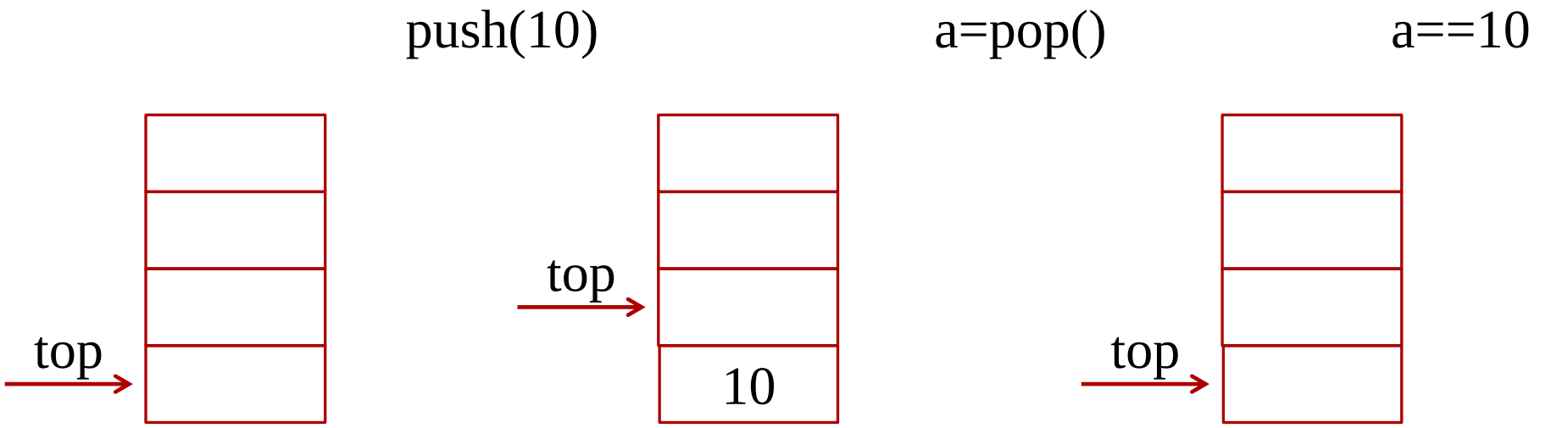
Abstractions

Abstraction v Implementation

- List abstraction
  - Grows and shrinks – insert, remove
  - Index into – lst[n]
- Stack abstraction (plate stack)
  - Push, pop
  - Simpler than list, as only access ‘top’
- Many possible implementations of each abstraction
  - Trade-offs

Stack Operations

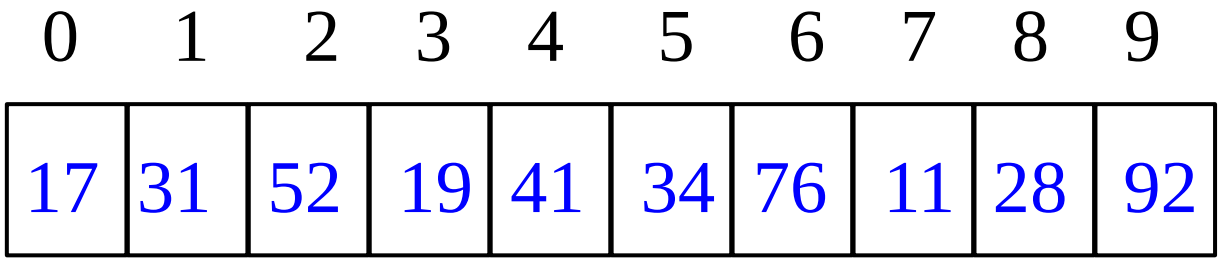
- pop and push



Exercise 1.2

- Operations on a stack

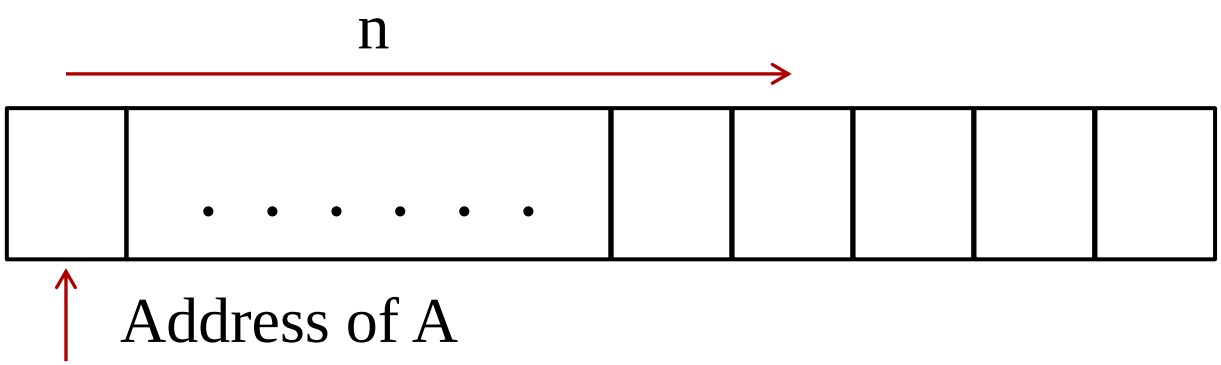
Lists in Arrays



- The array cannot grow
- To insert, we have to shuffle along
- Indexing is quick

- *Because indexing is quick, there are implementations based on arrays in lists*

Aside: Array and LMC

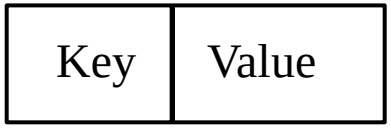


- Address of array – A – is address of entry 0
- Address of A[n] is is A + n

- Real computers
  - Have registers for addresses
  - Do arithmetic on addresses

Set & Map Abstractions

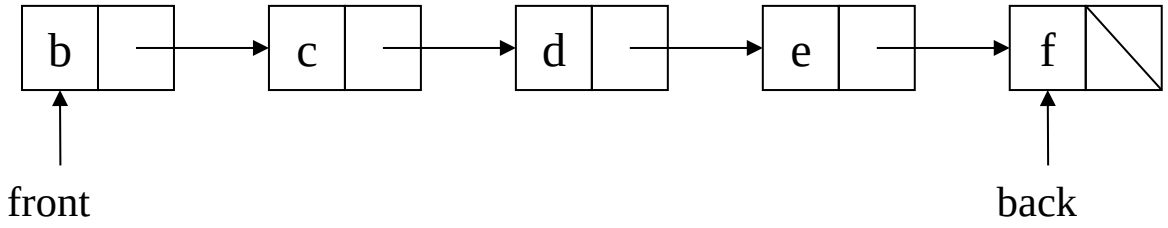
- Set
  - Membership
  - Insert
  - Remove
- Dictionary (Map)
  - Keys are like a set – value attached to each key
  - ... Set operations
  - Lookup value is similar to membership



Linked Lists

An implementation of the list abstractions

Linked List Concept



- Each entry has
  - A value
  - A pointer to the next entry
- Keep a pointer to the front entry
- The pointer of the last entry is None

Linked List Index

```
myList.index(i)
```

- Count along the list to entry i
- Return the value

```
pointer = front
count = 0
while count < i:
   pointer ← next of current entry
   count = count + 1
return the value of current entry
```

Linked List Update

```
myList.update(idx, newValue)
```

- Count along the list to entry index
- Replace value with new value

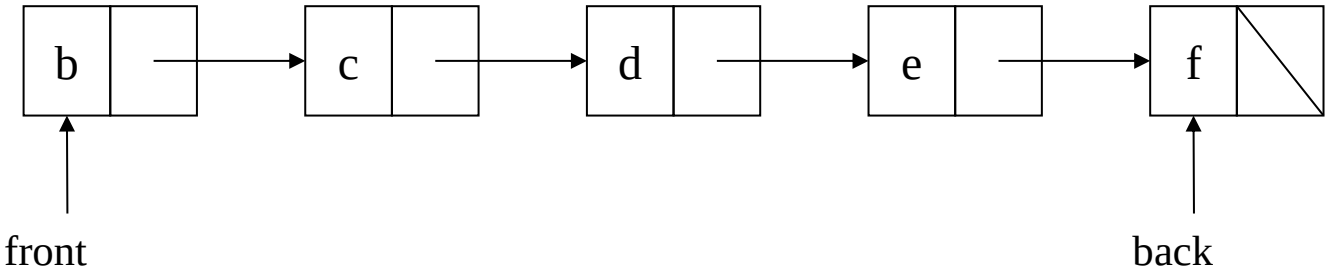
```
pointer = front
count = 0
while count < idx:
   pointer ← next of currentEntry
   count = count = 1
currentEntry.value = newValue
```

Linked List Insert

```
myList.insert(idx, newValue)
```

- Count along the list to entry idx-1
- Insert a new entry
  - Next pointer of current entry points to new entry
  - Next pointer of new entry points to following entry

Exercise



- Redraw list after:
  - appending a new entry at the end
  - inserting before entry zero
  - inserting before entry 3

Exercises 2.1 – 2.3

- Linked list

Linked List Code

```
class Entry:
   def __init__(self, v):
      self.value = v
      self.next = None

   def setValue(self, v):
      self.value = v
   def getValue(self):
       return self.value

   def setNext(self, n):
      self.next = n
   def getNext(self):
      return self.next
```

```
class List:
   def __init__(self):
      self.length = 0
      self.first = None

   def append(self, value):
      entry = Entry(value)
      if self.first == None :
         self.first = entry
         return
      p = self.first
      q = p.getNext()
      while q != None:
         p = q
         q = p.getNext()
      p.setNext(entry)

   def index(self, i):
      count = 0
      p = self.first
      while count < i:
         p = p.getNext()
         count = count + 1
      return p.getValue()
```

Complexity of Linked List

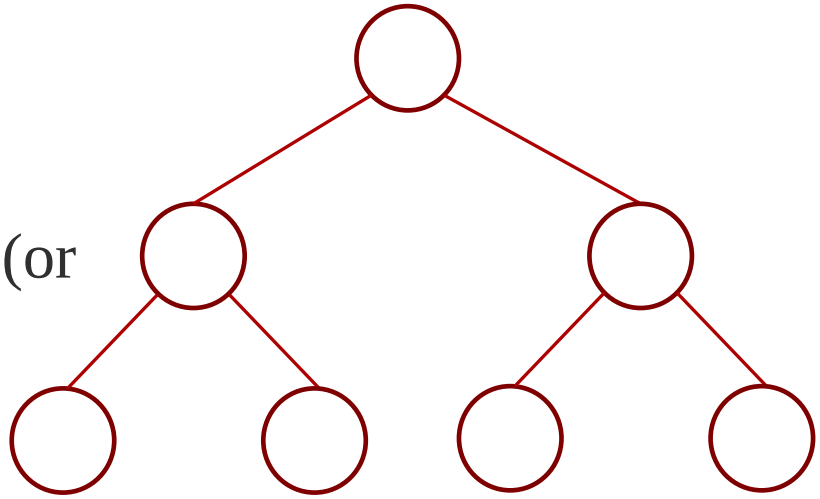
- Indexing is linear: O(n)
  - c.f. array index is O(1)
  - need to do better!
- Often need to visit every entry in the list
  - e.g. sum, **search**
  - This is $O(n^2)$ if we use indexing
  - Easy to improve this by keeping track of place in list
- Search is O(n)

Binary Trees

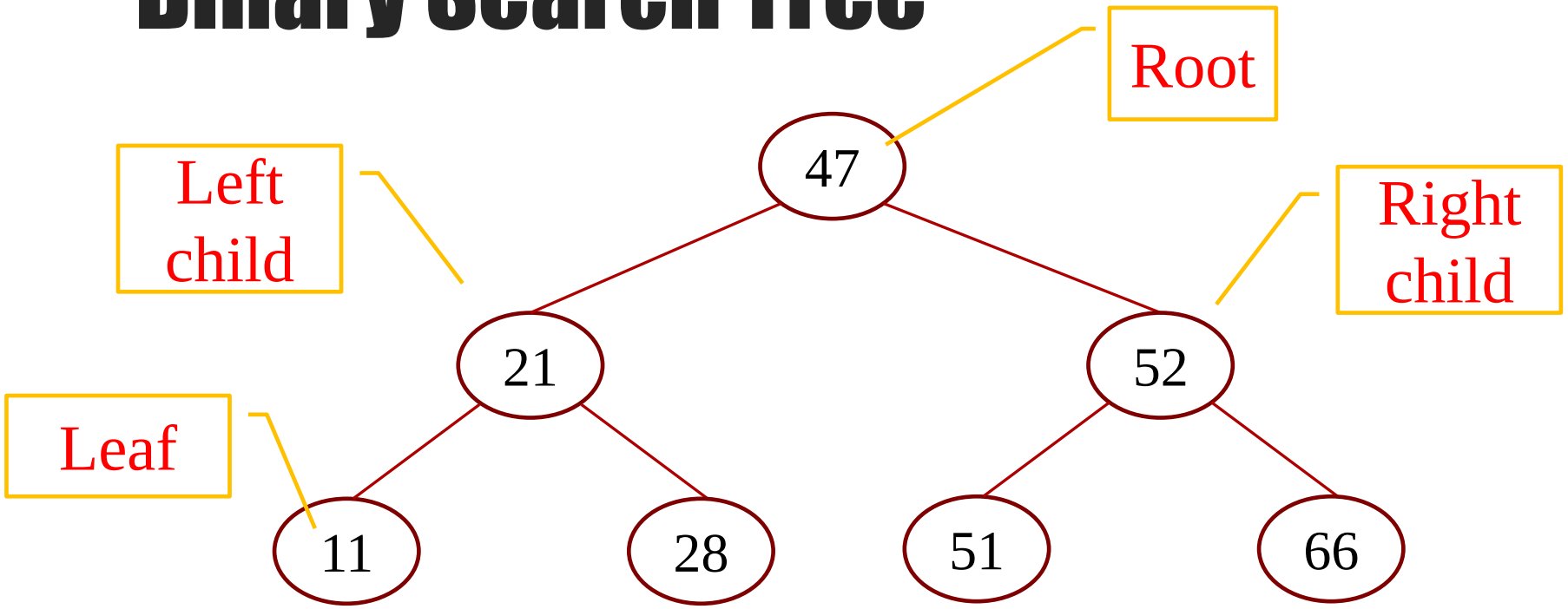
A more complex linked structure

Introduction

- Many uses of (binary) tree
- Key ideas
  - Linked data structure with 2 (or more) links (c.f. linked list)
  - Rules for organising tree
- Binary search tree
- Other uses
  - Heaps
  - Syntax trees

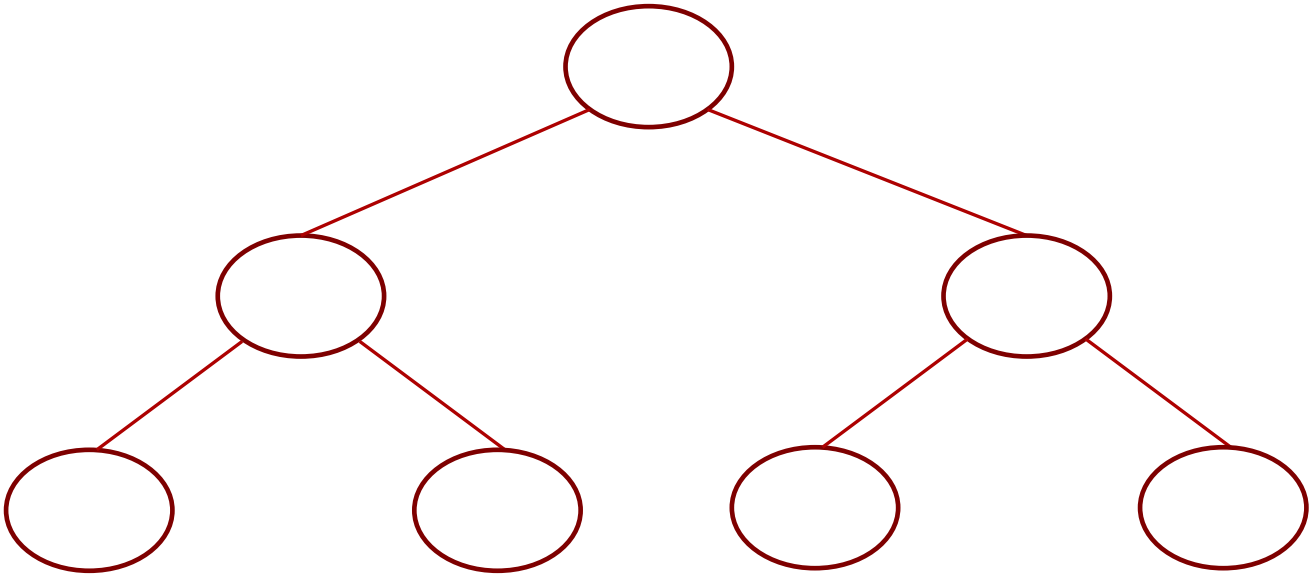


Binary Search Tree



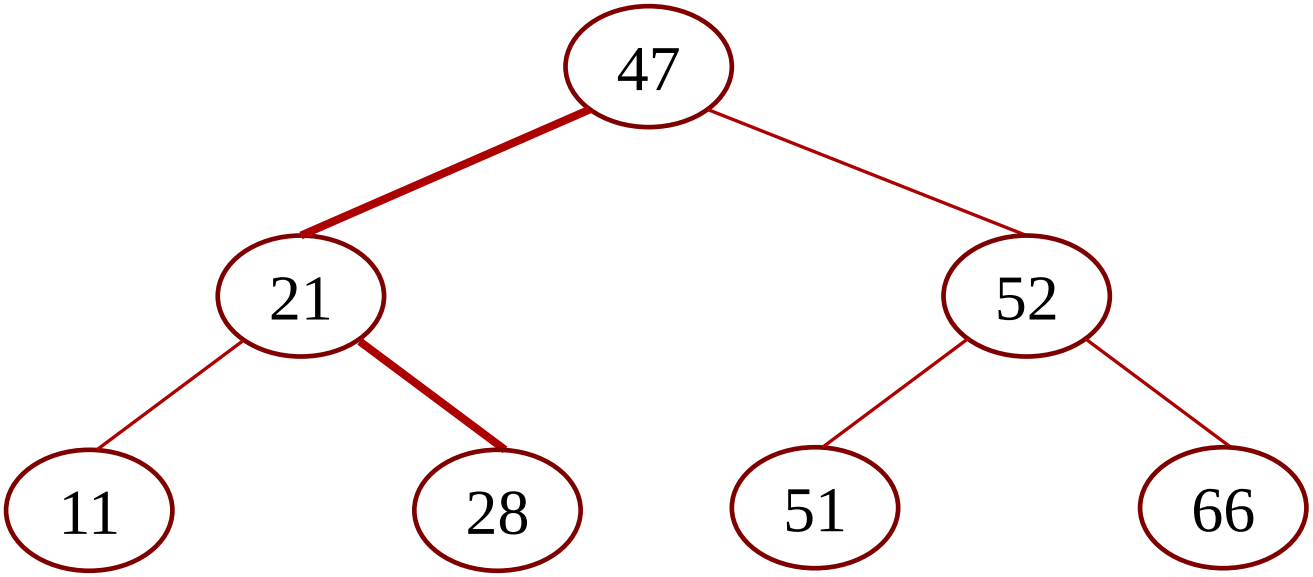
- All elements to the left of any node are < than all elements to the right

Exercise: Put The Element In



- 17, 19, 28, 33, 42, 45, 48

Search



- Binary search
- E.g. if target is 28:
  - 28 < 47 – go left
  - 28 > 21 – go right

What is the complexity of searching a binary tree?

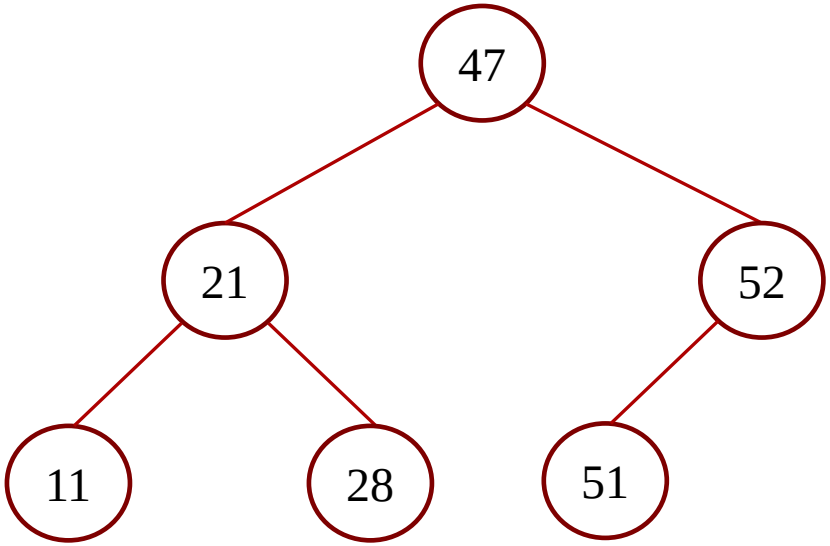
Binary Tree Search

- Recursive algorithms

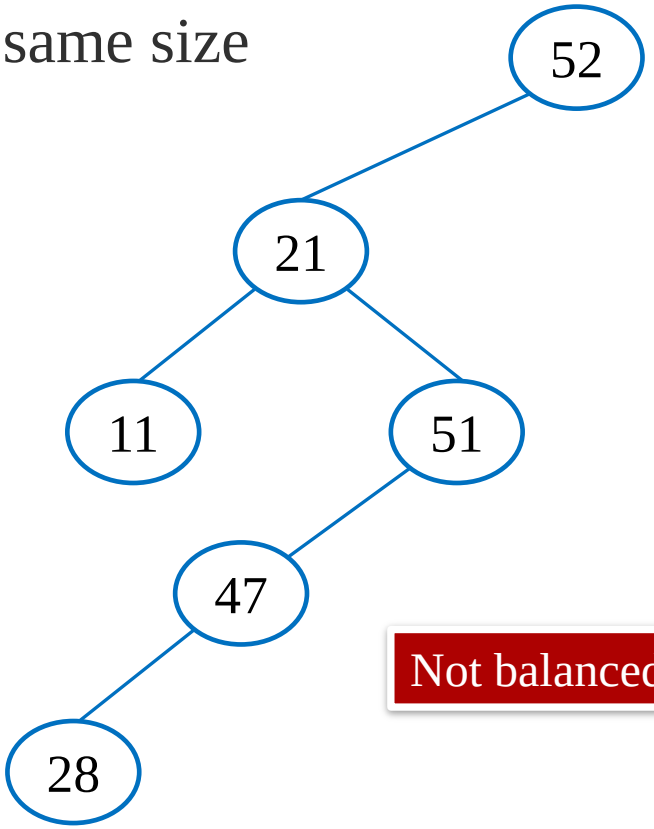
```
Find-recursive(key, node):  // call initially with node = root
    if node == None or node.key == key then
        return node
    else if key < node.key then
        return Find-recursive(key, node.left)
    else
        return Find-recursive(key, node.right)
```

Balance and Complexity

- Complexity depends on balance
  - Left and right sub-trees (nearly) same size
  - Tree no deeper than it need be



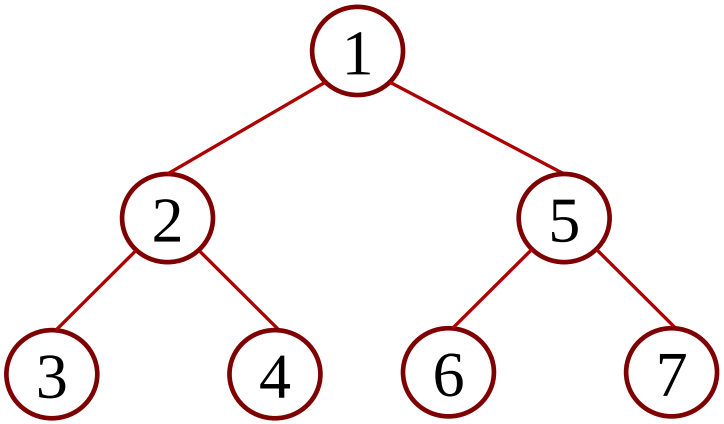
Balanced



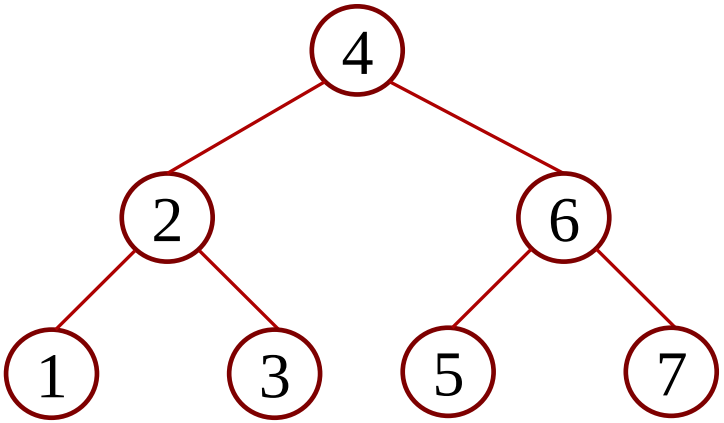
Not balanced

Tree Traversal

- Order of visiting nodes in tree



- In-order
  - Traverse the left subtree.
  - Visit the root.
  - Traverse the right subtree.



- Pre-order
  - Visit the root.
  - Traverse the left subtree.
  - Traverse the right subtree.

Exercises 3.1 – 3.3

- Binary trees

Hash Sets

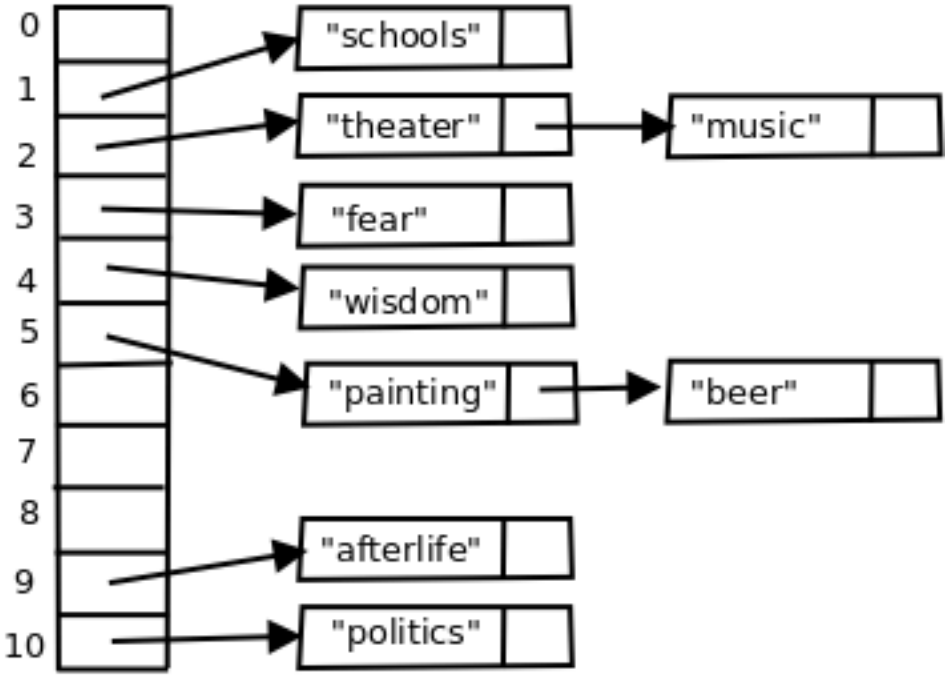
Insight

- Array are fast – indexing O(1)
- Map a string to an int, use as an array index
- hash() in Python
  - Any hashable type can be a dictionary key

- Ideally, should spread strings out

Hash Table (Set)

- Look for string S in array at:
  - hash(S) % size
- Collision
  - Two items share a hash
  - Linked list of items with same size
- Array length > number of items
- Array sized increased if table



Many Other Uses of Hashing

- Cryptography
- Checking downloads – checksum
- Checking for changes

Exercise

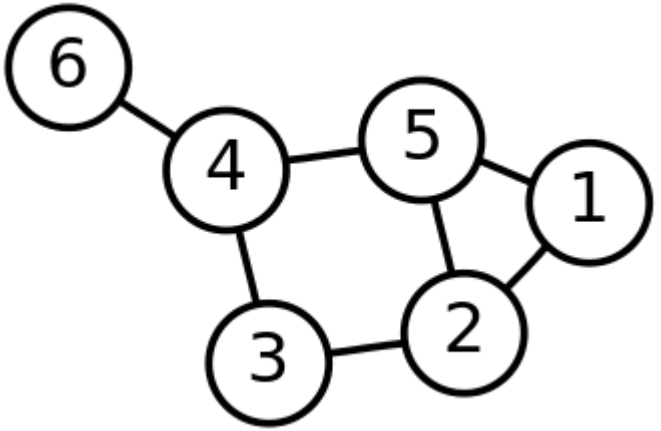
- Try Python hash function on strings and on other values

- Can you hash e.g. a person object?

Graphs

Graph

- Many problems can be represented by graphs
  - Network connections
  - Web links
  - ...
- Graphs have
  - Nodes and edges



- Edges can be directed or undirected
- Edges can be labelled

Graph v Trees

- Only one path between two nodes in a tree
- Graph may have
  - Many paths
  - Cycles (loops)

Graph Traversal

- Depth first traversal
  - Visit children,
  - ... then siblings
- Breadth first traversal
  - Visit siblings before children

- Algorithms similar to trees traversal, but harder (because of cycles)

Graph Algorithms

- Shortest paths
  - Final short path through graph with not-negative edge weight
  - Routing in networks
  - Polynomial
- Longest paths – travelling salesman
  - Intractable

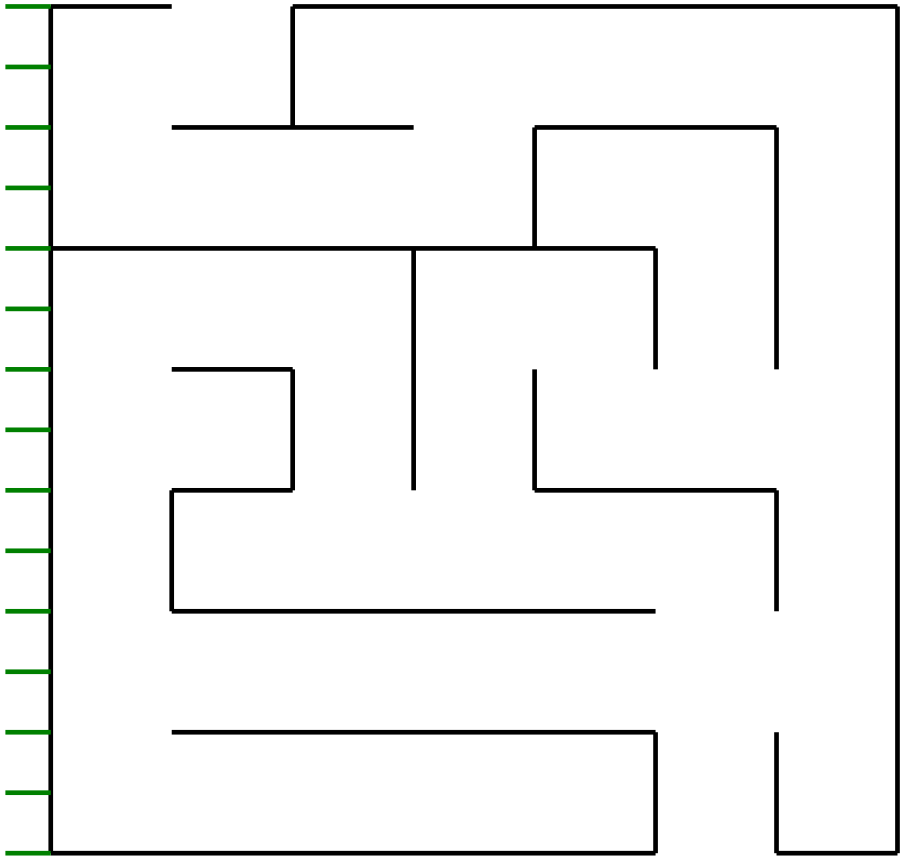
Graph Representations

- 2-D table of weights

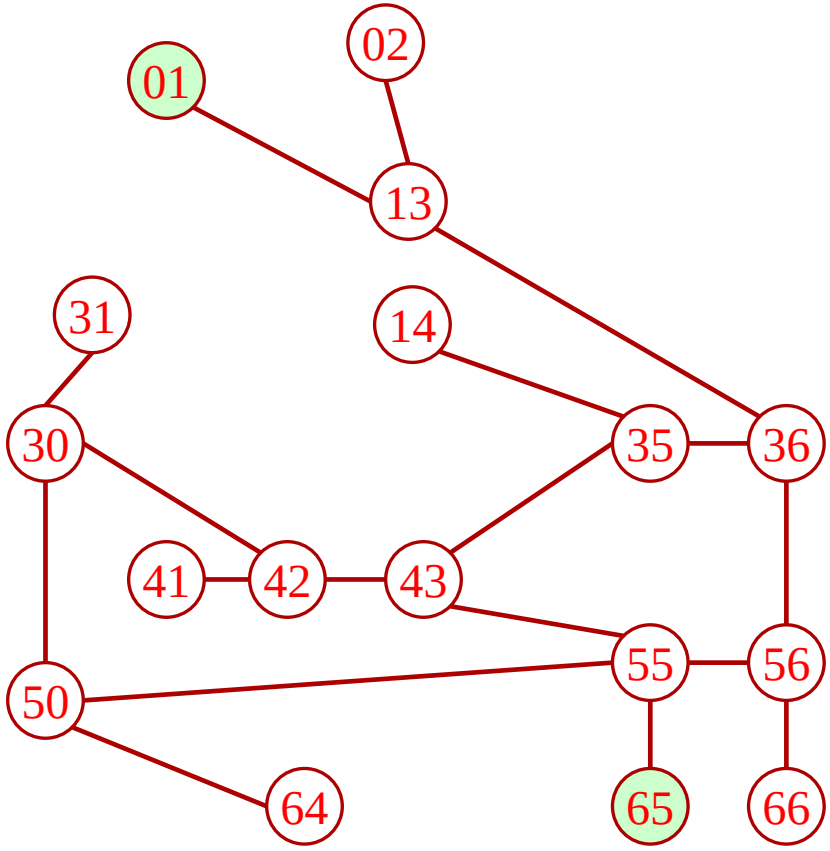
- Adjacency list
  - 1-D array of lists of neighbours

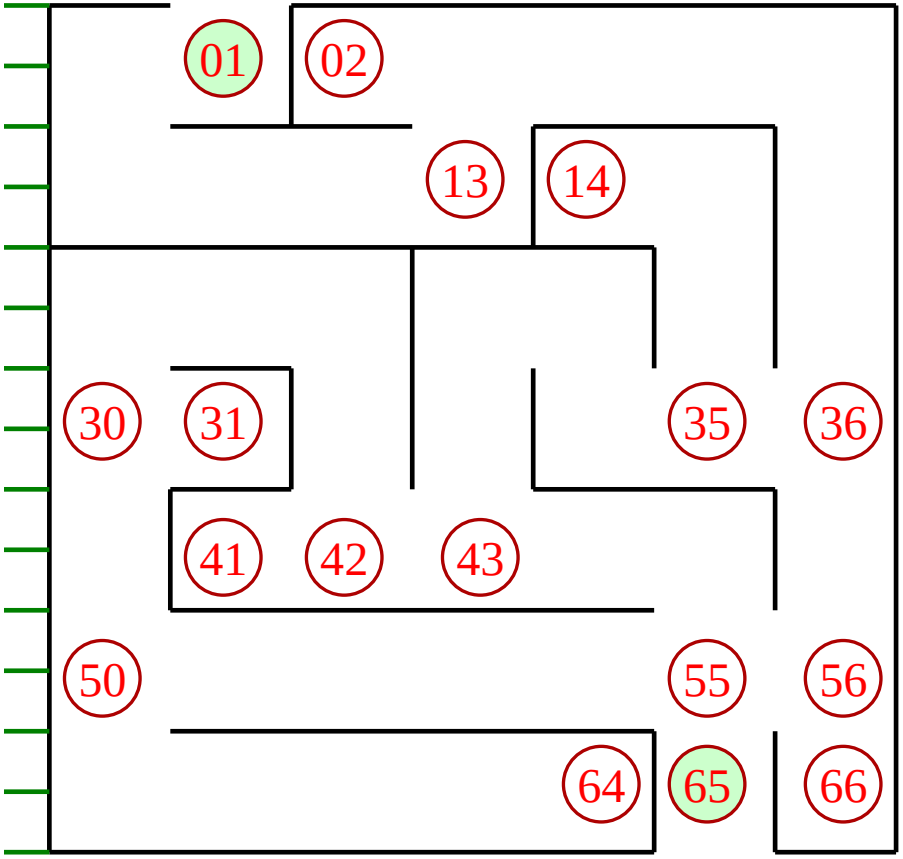
Exercise 4.1

- Show how maze can represented by a graph
- Maze solved by finding (shortest) path from start to finish



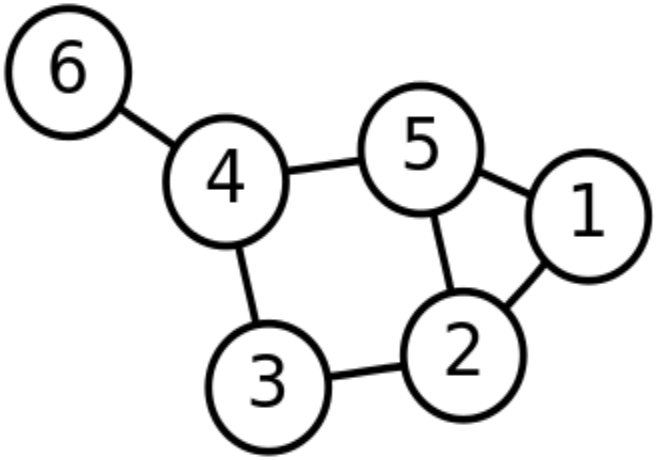
Exercise 4.1


01
02
13
31
14
30
35
36
41
42
43
55
56
50
64
65
66


01
02
13
14
30
31
35
36
41
42
43
50
55
56
64
65
66

Exercise 5.2

- Represent graph as
  - Adjacency list
  - A 2-D array (table)



Summary

- Python has lists and dictionaries built in
- Data structures to implement these
  - Linked data structures
  - Hashing: magic