



SIT320

Advanced Algorithms

Learning Summary Report

Jack Guan
221393284

Self-Assessment Details

The following checklists provide an overview of my self-assessment for this unit.

	Pass (D)	Credit (C)	Distinction (B)	High Distinction (A)
Self-Assessment	✓	✓	✓	

Self-Assessment Statement

	Included
Learning Summary Report	✓
Pass tasks complete	✓

Minimum Pass Checklist

	Included
All Credit Tasks are Complete on Doubtfire	✓

Minimum Credit Checklist (in addition to Pass Checklist)

	Included
Distinction tasks (other than Custom Program) are Complete	✓
Custom program meets Distinction criteria	✓

Minimum Distinction Checklist (in addition to Credit Checklist)

	Included
Something Awesome included	✓
Custom project meets HD requirements	

Minimum High Distinction Checklist (in addition to Distinction Checklist)

Declaration

I declare that this portfolio is my individual work. I have not copied from any other student's work or from any other source except where due acknowledgment is made explicitly in the text, nor has any part of this submission been written for me by another person.

Signature: **Jack Guan**

Jack Guan
221393284

Portfolio Overview

This portfolio includes work that demonstrates that I have achieved all Unit Learning Outcomes for SIT320 Unit Title to a **Distinction** level.

ULO1 — Advanced analysis: I compared theoretical complexity with empirical runtime and memory profiles across varied inputs statistical summary of results included.

ULO2 — Strategic design: I adapted and combined paradigms (e.g., DP + heuristic) to improve average-case performance on a hard problem, with proof/argument of correctness.

ULO3 — Efficient implementation: Implementations are production-quality, modular, tested, optimized data structures and include edge-case tests and profiling.

ULO4 — Critical evaluation: For each major task, I compared alternative trade-offs in time, memory, and practicality and justified the chosen approach.

I empirically validated the algorithmic complexity, designed improved algorithms to improve actual performance, and optimized well-tested code. In some projects, we compared the Dijkstra and A* algorithms under different constraints, selected the A* algorithm with domain-specific heuristics, and demonstrated significant improvements in runtime and memory efficiency on real-world datasets. These achievements demonstrate a comprehensive understanding of the algorithm and informed decision-making.

Reflection

The most important things I learnt:

The biggest lesson I gained from this unit is not a single algorithm, but a way of thinking — breaking down complex problems into structured components. For example, in the 6. Dynamic Programming task, I saw how dynamic programming turns a messy problem into a clear solution. This shifted my mindset from “how to code” to “how to model and abstract.”

The things that helped me most were:

Complexity analysis trained me to think about efficiency under worst, average, and amortized cases.

NP-completeness discussions showed me the limits of computation and when approximation is necessary.

I found the following topics particularly challenging:

Defining states in dynamic programming was hard, especially for non-standard problems. Proving correctness of advanced graph algorithms (e.g., Ford-Fulkerson) required extra time and effort.

I found the following topics particularly interesting:

Approximation algorithms showing practical ways to handle NP-hard problems.

I feel I learnt these topics, concepts, and/or tools really well:

I became confident with greedy design and graph traversal. For example, in 7. Greedy Algorithms task, I proved a greedy strategy's optimality; in 5. Graphs II task, I applied BFS/DFS effectively.

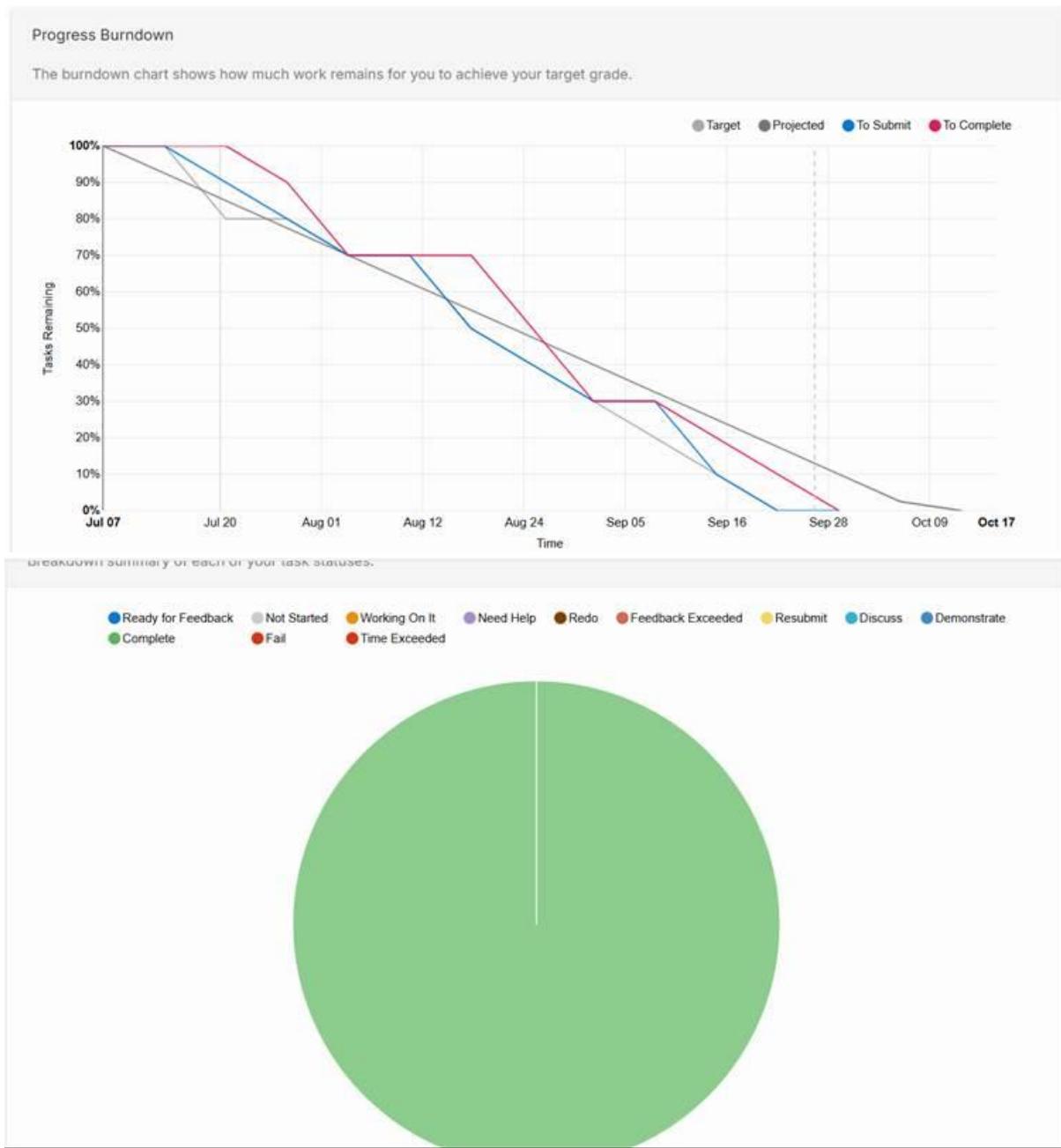
I still need to work on the following areas:

Applying advanced data structures Fibonacci heaps, segment trees for deeper optimization.

Practicing problem reductions to NP-complete problems more fluently.

My progress in this unit was ...:

My progress slowed down mid-semester when I learned about dynamic programming and NP-completeness problems, but I caught up by solidifying these concepts in my final project.



This unit will help me in the future:

This unit prepares me for technical interviews, postgraduate research, and high-performance system design. More importantly, it gives me confidence to face complex, real-world problems with algorithmic thinking.

If I did this unit again I would do the following things differently:

Form a study group earlier.

Build a personal “algorithm code library.”

Seek help sooner when stuck.

Other...:

I would say it is a wonderful study period in Australia, even though I'm no longer there currently. This is my final unit in Bachelor of Computer Science course, it takes me nearly 4 years to finish it, due to some unavoidable factors, Visa is one part of it. This is also a factor that prevents me from applying for a master's degree (at least now). But at least it left me with good memories, a life experience of Australia. 😊

DEAKIN UNIVERSITY

ADVANCED ALGORITHMS

YI GUAN

Portfolio Submission

Submitted By:

Yi GUAN
ppv

Tutor:

Nayyar ZAIDI

September 26, 2025



Contents

1 Learning Summary Report	1
2 Overall Task Status	2
3 Learning Outcomes	3
4 Introduction	4
5 Advanced Trees	26
6 Advanced Hashing and Sorting	109
7 Advanced Algorithmic Complexity	140
8 Graphs II	151
9 Dynamic Programming	188
10 Greedy Algorithms	211
11 Linear Programming	233
12 Network Flow Algorithms	273
13 AI Algorithms	287

2 Overall Task Status

Task	Status	Times Assessed
Introduction	Complete	1
Advanced Trees	Complete	3
Advanced Hashing and Sorting	Complete	2
Advanced Algorithmic Complexity	Complete	1
Graphs II	Complete	1
Dynamic Programming	Complete	1
Greedy Algorithms	Complete	1
Linear Programming	Complete	2
Network Flow Algorithms	Complete	1
AI Algorithms	Complete	1
Task 11 Advanced Algorithms	Not Started	

3 Learning Outcomes

4 Introduction

Introduction

Date	Author	Comment
2025/07/20 14:38	Yi Guan	Ready to Mark
2025/07/20 14:38	Yi Guan	hihi
2025/07/22 21:24	Xiangwen Yang	Well done! Please come to discuss this during the workshop.
2025/07/22 21:24	Xiangwen Yang	Discuss
2025/07/23 01:34	Yi Guan	cool, got it Sir
2025/07/25 20:23	Xiangwen Yang	Good work!
2025/07/25 20:24	Xiangwen Yang	Complete

DEAKIN UNIVERSITY

ADVANCED ALGORITHMS

ONTRACK SUBMISSION

Introduction

Submitted By:

Yi GUAN

ppv

2025/07/20 14:38

Tutor:

Xiangwen YANG

July 20, 2025



Student Details

Name: Jack Guan
Deakin Student ID: 221393284
Tutor: Dr.Nayyar Zaidi , Dr. Ziwei Hou , Mr Xiangwane Yang
Class: Online
Intended Grade: Distinction

Instructions: Please fill in the module name, along with submission and discussion deadlines from Ontrack website.

I will adhere to following timetable for submitting tasks, and will come to class for task discussion with my tutor.

SIT320 - Time Table

Module	Tasks	Submission Deadline	Discussion Deadline
1 Introduction	Must	Module 1 Task	20 July
2 Advanced Trees	Must	20 July	27July
3 Advanced Hashing Sorting	Must	3 Aug	10 Aug
4 Advanced Algorithmic Complexity	Must	17 Aug	24 Aug
5 Graphs II	Must	17 Aug	24 Aug
6 Dynamic Programming	Must	17 Aug	24 Aug
7 Greedy Algorithms	Must	24 Aug	31 Aug
8 Linear Programming	Must	31 Aug	7 Sep
9 Network Flow Algorithms	Must	7 Sep	14 Sep
10 AI Algorithms	Must	14 Sep	21 Sep
11 Task 11 Advanced Algorithms		21 Sep	28 Sep

Discussed with your Tutor (Circle one): Yes / No

Signatures: _____ Jack Guan _____

Task 1: Lesson Review

Write a one-page reflection summarising what you have learned in this module.

This unit reveals how algorithms are more broadly problem solving. This blends bottom-up and top-down perspectives — using tools like pseudocode and Python. Here are the key takeaways:

Algorithm Design and Problem Solving

The bottom-up/top-down distinction. Bottom-up views algorithms as input-output processes; top-down embeds them in the Polya cycle (understand → plan → solve → check). Techniques like dynamic programming are derived from mathematics and applied to real-world problems. As Siggi points out, experience will hone this in areas like data science, where data structures support rather than dictate solutions.

Pseudocode: A Strategic Bridge

Pseudocode avoids syntax distractions and enables language-agnostic design. Good form prioritizes readability, modularity, and ease of conversion; it serves a dual purpose: team collaboration and upfront debugging. Writing the logic first can catch bugs that the compiler misses. Adopting Python-style pseudocode will streamline my workflow, ensuring modular decomposition and efficient peer review.

Python and Jupyter: The Power of Prototyping

Its pseudocode-like syntax makes the transition from design to code more fluid. Jupyter's cell-based execution supports iterative Polya loops: test steps, visualize results, and

optimize - all in one space. Use it to tackle data science challenges and merge prototyping, visualization, and documentation.

The main content of this module is: top-down problem framing guides algorithm selection, pseudocode clarifies logic, and Python/Jupyter accelerates iteration.

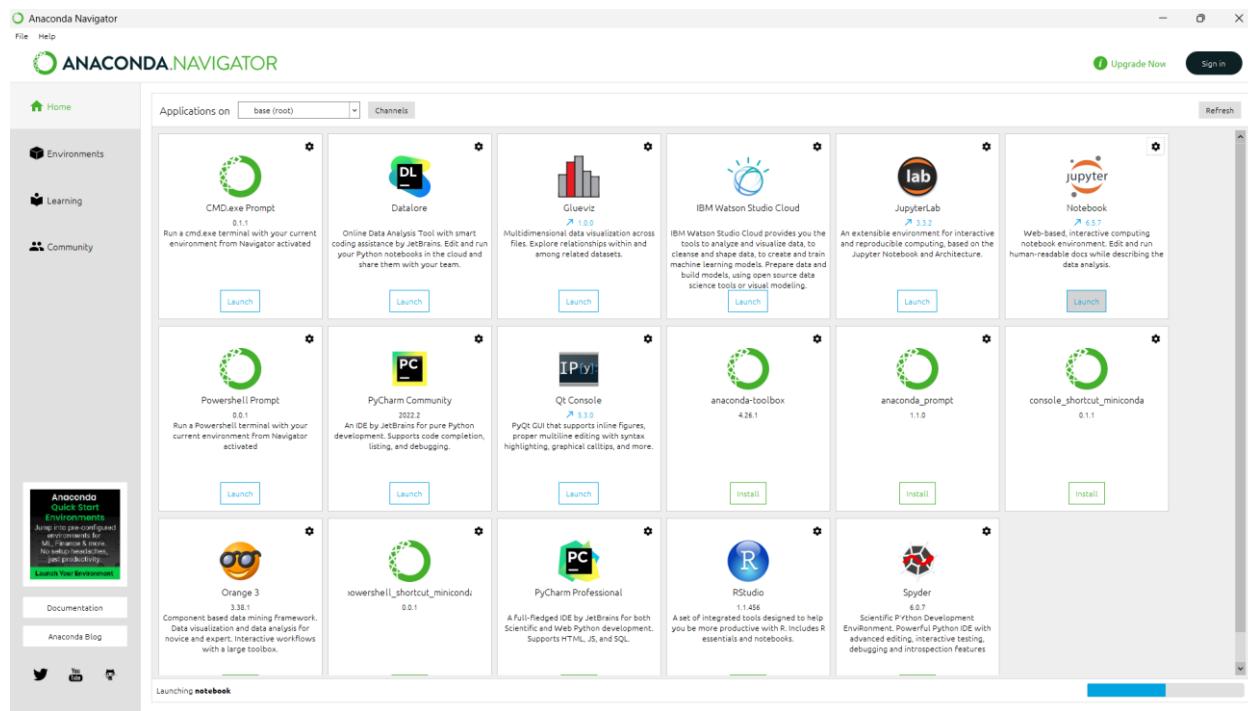
Task 4: Python Environment Setup

You must set up your Python environment and demonstrate basic programming ability.

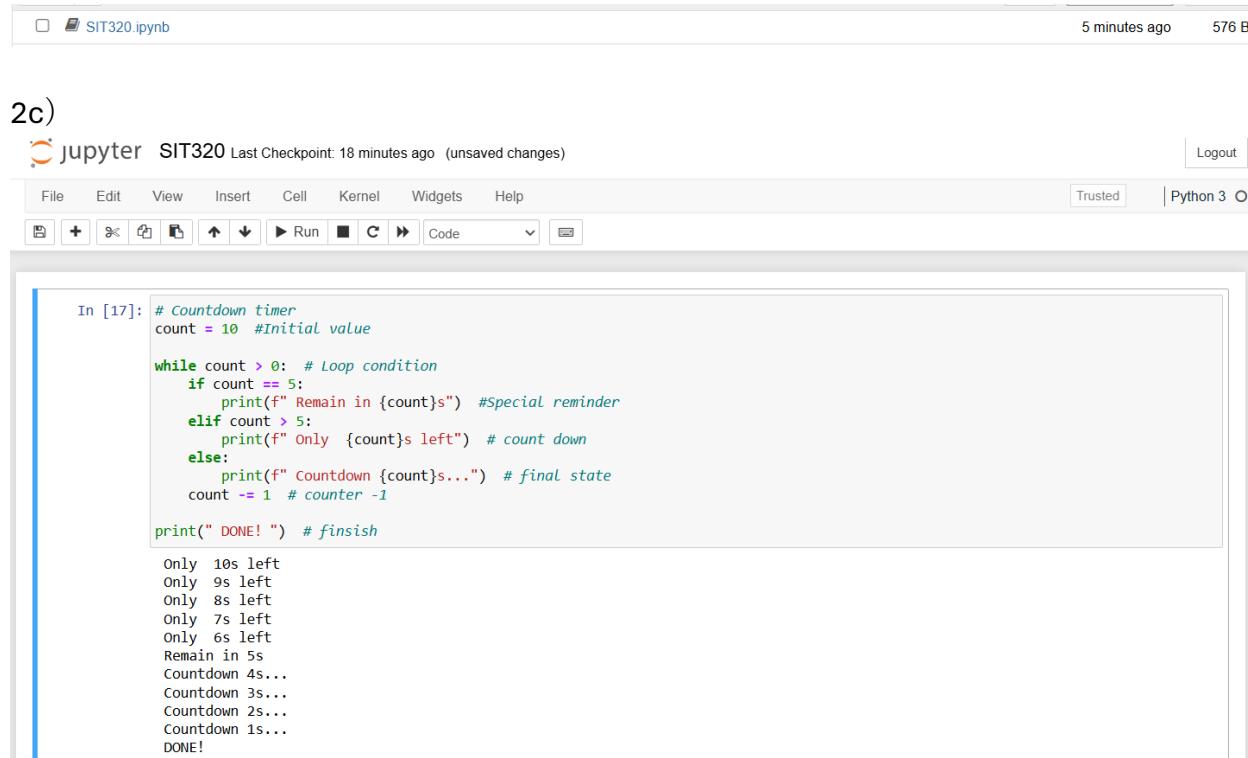
- (2a) Provide evidence that Anaconda Navigator is installed on your system.
- (2b) Show that you have created a new environment named SIT320.
- (2c) Demonstrate execution of a basic Python script using while and if statements.

Screenshots or screen recordings are acceptable.

2a)



2b)



While loop:

Repeat the indented code block when count > 0

Execute count -= 1 after each iteration (counter)

If loop:

if/elif/else conditional branch

count == 5: Special reminder

count > 5: Count down

else: Final state

When value is < 0 , end loop.

Task 5: Complexity Analysis of Your Algorithm

Analyze the time complexity of your Tic-Tac-Toe algorithm. Is it a polynomial-time algorithm (in P), or does it fall under NP?

SIT320 – Advanced Algorithms

Your response should reflect a clear understanding of algorithmic complexity and the difference between P and NP problems. Your tutor will discuss this with you.

Algorithm complexity:

Time complexity: $O(9)$ (worst case)

In practical applications, the amount of search can be reduced by pruning.

Space complexity: $O(9)$

The recursive depth is 9.

Tic-tac-toe is a Polynomial-time algorithm problem because its solution process can be completed through a deterministic polynomial time algorithm. The verification and search for the optimal solution can be carried out within a reasonable time frame. It cannot be other NP-complete problems.

Task 7: [Optional – For Distinction and High Distinction Students] Advanced Exploration

Analyze your algorithm that you designed for Tic-Tac-Toe, and critically analyze your strategy. Can you use your approach to design solution to games like Chess, backgammon? If not, what modifications will you make? This task will prepare you for you D and HD tasks, when we design a Reinforcement Learning based solution to Tic-tac-toe.

The core strategy of the pseudocode is to use the Minimax algorithm to simulate all subsequent games for each possible move, calculate the result score, backtrack and select the most favorable step for the current player.

The time complexity is constant (for a fixed 3x3 chessboard)

But it also has certain limitations, because it is not scalable and cannot be expanded to larger chessboards or complex games.

At the same time, the Minimax algorithm cannot automatically adapt to games with different rules and is less efficient.

It is unlikely to be achieved for chess because it has a huge state space and a deep game tree, and Minimax cannot be fully searched.

If certain changes are to be made, more powerful optimization techniques and modeling methods need to be introduced.

For example: set a search depth limit; use a heuristic evaluation function instead of a final judgment, etc.

1 Simple Algorithm to play a game of Tic-Tac-Toe

CELL 02

```
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:85% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

The code is downloaded from here: https://github.com/doctorsmonsters/minimax_tic_tac_toe/blob/main/minimax%20TicTacToe.ipynb I do not take any responsibility for its inner working.

Your task is to understand it, evaluate it, and make it working.

There can be problems in the code. Try fixing them, as part of setting-up the test cases activity

1.1 Board Definition

CELL 05

```
#define the board
board={1: ' ', 2: ' ', 3: ' ',
       4: ' ', 5: ' ', 6: ' ',
       7: ' ', 8: ' ', 9: ' '}
```

CELL 06

```
#function to print the board
def printBoard(board):
    print(board[1] + ' | ' + board[2] + ' | ' + board[3])
    print(' -+---')
    print(board[4] + ' | ' + board[5] + ' | ' + board[6])
    print(' -+---')
    print(board[7] + ' | ' + board[8] + ' | ' + board[9])
    print('\n')
```

CELL 07

```
printBoard(board)
```

```
| |  
-+--  
| |  
-+--  
| |
```

CELL 08

```
#function to check if a certain position in the board is empty.
def spaceIsFree(position):

    if (board[position]==' '):
        return True
    else:
        return False
```

CELL 09

```
#spaceIsFree(1)
#Illegal call outside a function
```

CELL 10

```
#method to insert letter in space
def insertLetter(letter, position):

    if (spaceIsFree(position)):
        board[position]=letter
        printBoard(board)

    #added in main function
    # if (chkDraw()):
    #     print('Draw!')
    # elif (chkForWin(letter)):
    #     if (letter=='X'):
    #         print('Bot wins!')
    #     else:
    #         print('You win!')
    #     return True
    return False

#else:
#    print('Position taken, please pick a different position.')
#    position=int(input('Enter new position: '))
#    insertLetter(letter, position)

# return
```

CELL 11

```
#function to check if board is draw
def chkDraw():

    for key in board.keys():
        if (board[key]==' '):
            return False
    return True

#function to check if one user has won
#Does not take any parameters to check the state of a certain player,
#should make sure it works.
def chkForWin(player):

    if (board[1]==board[2] and board[1]==board[3] and board[1] !=' '):
        return True
    elif (board[4]==board[5] and board[4]==board[6] and board[4] !=' '):
        return True
    elif (board[7]==board[8] and board[7]==board[9] and board[7] !=' '):
        return True
    elif (board[1]==board[5] and board[1]==board[9] and board[1] !=' '):
        return True
```

```
    return True
elif (board[3]==board[5] and board[3]==board[7] and board[3] != ' '):
    return True
elif (board[1]==board[4] and board[1]==board[7] and board[1] != ' '):
    return True
elif (board[2]==board[5] and board[2]==board[8] and board[2] != ' '):
    return True
elif (board[3]==board[6] and board[3]==board[9] and board[3] != ' '):
    return True
else:
    return False
```

CELL 12

```
#function to check who won
def chkMarkForWin(mark):

    if (board[1]==board[2] and board[1]==board[3] and board[1] ==mark):
        return True
    elif (board[4]==board[5] and board[4]==board[6] and board[4] ==mark):
        return True
    elif (board[7]==board[8] and board[7]==board[9] and board[7] ==mark):
        return True
    elif (board[1]==board[5] and board[1]==board[9] and board[1] ==mark):
        return True
    elif (board[3]==board[5] and board[3]==board[7] and board[3] ==mark):
        return True
    elif (board[1]==board[4] and board[1]==board[7] and board[1] ==mark):
        return True
    elif (board[2]==board[5] and board[2]==board[8] and board[2] ==mark):
        return True
    elif (board[3]==board[6] and board[3]==board[9] and board[3] ==mark):
        return True
    else:
        return False
```

CELL 13

```
player = '0'
bot = 'X'
```

CELL 14

```
#function for player move
def playerMove():
    while True:
        try:
            position = int(input('Enter position for 0 (1-9): '))
            if position < 1 or position > 9:
                print("Please enter a position between 1 and 9.")
            elif spaceIsFree(position):
                insertLetter('0', position)
                return
            else:
                print("Position already taken!")
        except ValueError:
            print("Invalid input, please enter a number.")

#function for bot move
def compMove():

    bestScore=-1000 #lowest to start with, for comparison to the score after a certain move

    bestMove=0

    for key in board.keys(): #for loop to find empty slots, make a move, calculate the score and see if its higher then best score
        if (board[key]==' '):
            board[key]=bot
            score = minimax(board, 0, False) #calculate the score
            board[key] = ' ' #set board back to what it was
            if (score > bestScore): #compare the score from a certain move to the best score.
                bestScore = score
                bestMove = key
```

```
#at the end of the loop, we have the best move figured out.  
insertLetter(bot, bestMove)  
return
```

CELL 15

```

def minimax(board, depth, isMaximizing):

    if (chkForWin(bot)):
        return 1
    elif (chkForWin(player)):
        return -1
    elif (chkDraw()):
        return 0

    if isMaximizing:

        bestScore = -1000

        for key in board.keys():
            if board[key]==' ':
                board[key]=bot
                #Depth parameter increment error, the value of depth is always 0.
                #score = minimax(board, 0, False)
                score = minimax(board, depth + 1, False)
                board[key]= ' '
                if (score>bestScore):
                    bestScore = score
        return bestScore

    else:

        bestScore = 1000

        for key in board.keys():
            if board[key]==' ':
                board[key]=player
                #Depth parameter increment error, the value of depth is always 0.
                #score = minimax(board, 0, True)
                score = minimax(board, depth + 1, False)
                board[key]= ' '
                if (score<bestScore):
                    bestScore = score
        return bestScore

```

CELL 16

```

#Not correctly determining whether the game is over.
#while not chkForWin(player):
#while not chkForWin('X') and not chkForWin('O') and not chkDraw():
#    compMove()
#    playerMove()
def main():
    printBoard(board)
    while True:
        compMove()
        if chkForWin(bot):
            print('Bot wins!')
            break
        elif chkDraw():
            print('Draw!')
            break

        playerMove()
        if chkForWin(player):
            print('You win!')
            break
        elif chkDraw():
            print('Draw!')

```

```
        break

if __name__ == "__main__":
    main()
```

```
| |  
-+--  
| |  
-+--  
| |
```

```
X| |  
-+--  
| |  
-+--  
| |
```

```
Enter position for O (1-9): 2
```

```
X|O|  
-+--  
| |  
-+--  
| |
```

```
X|O|X  
-+--  
| |  
-+--  
| |
```

```
Enter position for O (1-9): 5
```

```
X|O|X  
-+--  
|O|  
-+--  
| |
```

```
X|O|X  
-+--  
X|O|  
-+--  
| |
```

```
Enter position for O (1-9): 7
```

```
X|O|X  
-+--  
X|O|  
-+--  
O| |
```

```
X|O|X  
-+--  
X|O|X  
-+--  
O| |
```

```
Enter position for O (1-9): 9
```

```
X|O|X  
-+--  
X|O|X  
-+--  
O| |O
```

```
X|O|X  
-++-  
X|O|X  
-++-  
O|X|O
```

Draw!

	CELL 17
3	

5 Advanced Trees

Advanced Trees

Date	Author	Comment
2025/07/20 14:39	Yi Guan	Ready to Mark
2025/07/20 14:39	Yi Guan	hihi
2025/07/25 20:25	Xiangwen Yang	Complete
2025/07/25 20:25	Xiangwen Yang	Fix and Resubmit
2025/07/25 20:28	Yi Guan	Ready to Mark
2025/07/25 20:28	Yi Guan	Hi Sir, here is the update for my code.
2025/07/25 20:32	Xiangwen Yang	Fix and Resubmit
2025/07/25 21:07	Yi Guan	:hot_face:Sir, I have requested extension for this task, I'll resubmit this task later tonight.
2025/07/25 21:27	Yi Guan	Ready to Mark
2025/07/25 21:27	Yi Guan	This time should be all correct. I wish
2025/07/30 09:23	Xiangwen Yang	Well done!
2025/07/30 09:23	Xiangwen Yang	Complete

DEAKIN UNIVERSITY

ADVANCED ALGORITHMS

ONTRACK SUBMISSION

Advanced Trees

Submitted By:

Yi GUAN

ppv

2025/07/25 21:27

Tutor:

Xiangwen YANG



July 25, 2025

Task 1: Lesson Review

Write a one-page reflection summarising what you have learned in this module.

This module provides a comprehensive review of binary search trees (BSTs) and their balanced variants. Trees are a special type of graph that are essential for implementing efficient insertion, deletion, and search (IDS) operations, ideally in $O(\log n)$ time complexity. BSTs embody this goal, but when unbalanced (e.g. when inserting nodes in ascending order), their time complexity drops to linear time ($O(n)$).

To alleviate this problem, self-balancing trees such as AVL trees and red-black trees (RBs) were developed. AVL trees have the property of maintaining height balance, ensuring that the height difference between the left and right subtrees of each node remains within ± 1 . When an insertion or deletion operation disrupts this balance, AVL trees employ tree rotations (single or double) to restore balance. While AVL trees are more strict in enforcing balance, they are faster to search due to their tighter height bounds.

On the other hand, red-black trees provide a less strict but more efficient balancing mechanism that associates each node with a color (red or black) and maintains five key properties (e.g., the root node is black, red nodes have black children, and all paths have the same number of black nodes). This more relaxed balancing mechanism simplifies insertion and deletion operations but slightly increases the height of the tree. Similar to AVL trees, red-black trees also use rotation operations, but these operations are selectively applied through internal rules.

In summary, this module highlights the trade-offs between search efficiency and update cost in different tree structures. Knowing when to use a basic binary search tree (BST) instead of a self-balancing variant such as an AVL tree or a red-black tree is critical to system performance.

Task 8: Compare Internal Node Structure

After constructing both a B Tree and a B+ Tree using the same keys (above):

- Count and compare the number of internal nodes.
- Which tree structure results in fewer internal nodes? Why?

How does this difference affect space usage and lookup performance in large-scale databases?

Comparison of the number of internal nodes: B-tree vs. B+ tree

B-tree and B+ tree are built using the same key set. B+ tree usually has fewer internal nodes than B-tree.

B-tree stores actual data directly in all nodes, while B+ tree stores data only in leaf nodes, and internal nodes only store key values for navigation. This structure makes the internal nodes of B+ tree more "lightweight", so that more keys can be stored.

Since the internal nodes of B+ tree do not need to be accompanied by data pointers, each node can accommodate more keys. This means that the same amount of data can be branched with fewer internal nodes in B+ tree, further reducing the height of the tree.

The structure of B+ tree is usually shorter than that of B tree, which not only reduces the traversal path, but also directly leads to fewer internal nodes required. Since the internal nodes of B-tree are responsible for data storage, their branching capacity is limited and more levels are required to cover the same data range.

Space efficiency: B+ tree has smaller internal nodes and takes up less space, which is suitable for database systems with large index scales and helps reduce overall storage costs.

Disk I/O performance: Smaller internal nodes mean that more nodes can be loaded into memory, reducing frequent access to the disk, thereby improving overall query performance.

In terms of range queries, the advantages of B+ tree are more obvious. Because its leaf nodes are connected by linked lists, data can be traversed sequentially without backtracking or re-searching the tree structure.

Task 9: Range Query in B+Tree

Using your B+ Tree from Task 6, design a range query function that returns all keys between two values, e.g., `range_query("cop", "max")`.

- Write a pseudocode for your algorithm.
- Explain how leaf-level pointers in a B+ Tree help support this functionality.

Page 2 of 2

First, since the linked list structure is formed between leaf nodes, the system can sequentially traverse all leaf nodes by accessing the key value in sequence without backtracking to the parent node or retraversing the entire tree. Once the starting point of the range query is found, the subsequent leaf nodes can be sequentially accessed through linear scanning until the query ends. The time complexity of this operation is closer to $O(n)$.

In addition, the physical layout of leaf nodes on the disk is usually continuous, which can significantly reduce disk I/O operations, make full use of the sequential reading characteristics of the disk, and avoid costly random access.

From an implementation perspective, the linked list structure of leaf nodes also simplifies the query logic. There is no need to design a complex tree traversal algorithm, and a simple linear traversal is required to complete the range query. This not only improves code efficiency, but also reduces the possibility of system errors.

More importantly, this linked list structure ensures that all key values are not missed and can fully cover the specified query range. When a key value greater than the query end point is encountered during the scan, the system can terminate the query in advance, further improving efficiency.

Therefore, the B+ tree achieves efficient, accurate and easy-to-implement range query capabilities through the ordered pointer chain of leaf nodes.

1 Lab: Trees

Lab associated with Module: Trees

CELL 04

```
# The following lines are used to increase the width of cells to utilize more space on the
# screen
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:95% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

1.0.1 Section 0: Imports

CELL 07

```
import numpy as np
```

Following libraries have to be installed on your computer. Try to install graphviz by using: conda install python-graphviz

I made use of some of the following links to get rid of errors:

<https://github.com/quadram-institute-bioscience/albatradis/issues/7>

<https://stackoverflow.com/questions/35064304/runtimeerror-make-sure-the-graphviz-executables-are-on-your-systems-path-aft>

<https://github.com/xflr6/graphviz/issues/68>

<https://github.com/RedaOps/ann-visualizer/issues/12>

On my mac computer I had to install some packages using brew to get rid of following error:
"ExecutableNotFound: failed to execute ['dot', '-Tsvg'], make sure the Graphviz executables are on your systems' PATH"

brew install graphviz

CELL 09

```
from IPython.display import Image
from graphviz import Digraph
from bisect import insort
```

Details of Digraph package: <https://h1ros.github.io/posts/introduction-to-graphviz-in-jupyter-notebook/>

1.0.2 Section 1: Testing Visualization Package

Let us test this visualization Digraph Pacakge, it is only a tool for displaying tree or graph, this will come handy as it helps to visualize our solution.

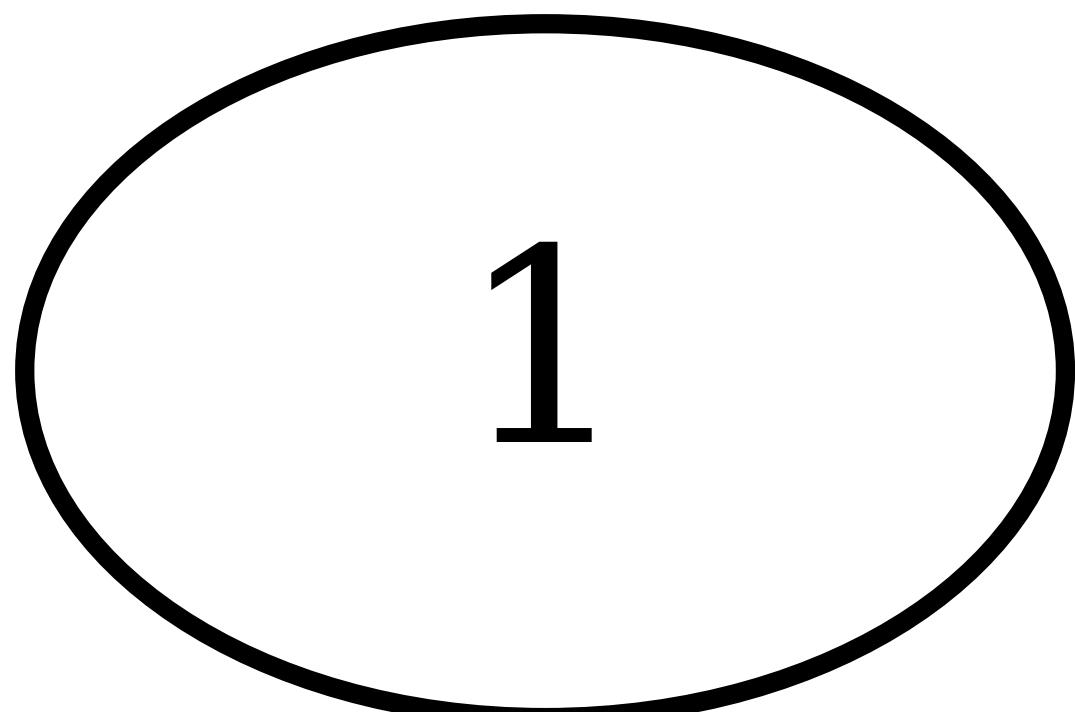
CELL 14

```
dot = Digraph()
dot.node("1")
dot.node("2")
dot.edges(['12'])
```

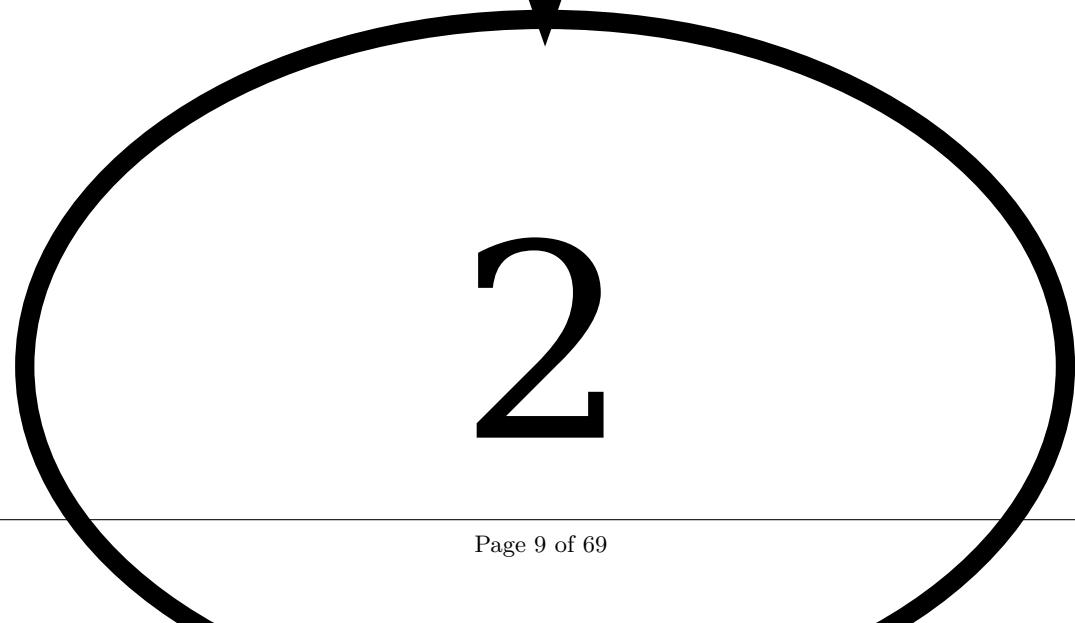
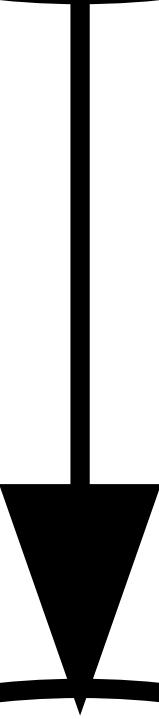
CELL 15

```
dot
```





1



2

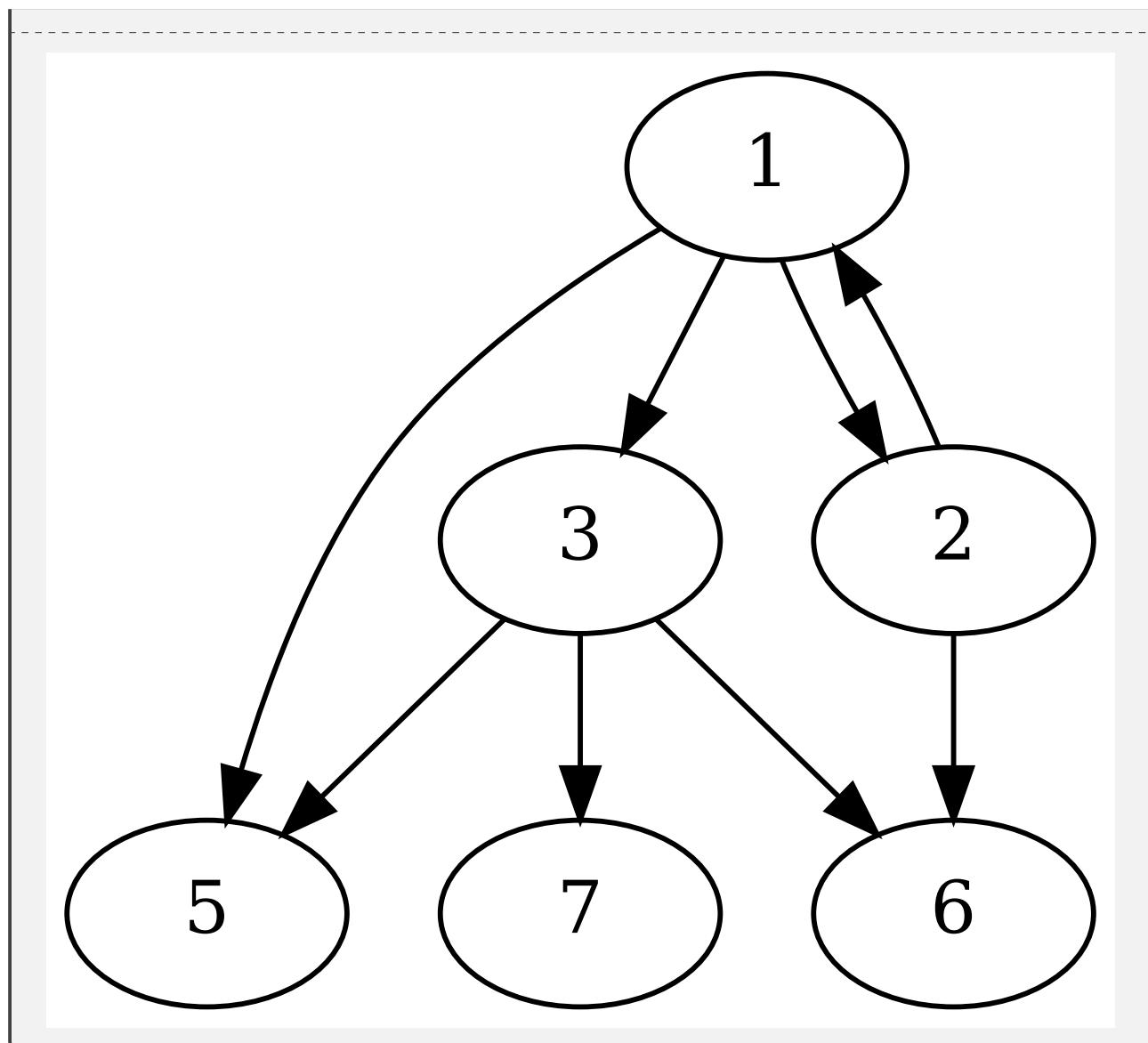
CELL 16

```
# Create Digraph object
dot = Digraph()

# Add nodes
dot.node('1')
dot.node('3')
dot.node('2')
dot.node('5')
dot.node('6')
dot.node('7')

# Add edges
dot.edges(['12', '13', '35', '15', '21', '37', '36', '26'])

# Visualize the graph
dot
```



1.0.3 Section 2: Creating a Binary Search Tree

Let us start by creating a BST

We will keep code simple in the sense that we will make a node class, and then build functions outside the class to implement various functionality.

CELL 21

```
class Node:

    def __init__(self, value):
        self.val = value
        self.right = None
        self.left = None
        self.parent = None
        self.balance = 0

def buildBinaryTree(nodes):

    if len(nodes) == 0:
        raise ValueError('list is empty')

    return binaryTree(nodes, 0, len(nodes) - 1)

def binaryTree(nodes, start, end):

    if start > end:
        return

    middle = (start + end) // 2
    root = Node(nodes[middle])
    root.left = binaryTree(nodes, start, middle - 1)
    root.right = binaryTree(nodes, middle + 1, end)

    return root
```

CELL 22

```
test1 = [1, 2, 3, 4, 5, 6, 7, 8]
test2 = [-1, 0, 9, 10]
```

CELL 23

```
test1_tree = buildBinaryTree(test1)
test2_tree = buildBinaryTree(test2)
```

We will make the simpler assumption that all the keys are unique when we are inserting

CELL 25

```
test3 = [0, 1, 2, 3, 3, 3, 5]
test3 = np.unique(test3)
```

CELL 26

```
test3_tree = buildBinaryTree(test3)
```

Okay now that we have build three trees, let us visualize them. For visualization, we will have to write another function.

CELL 28

```
def visualize_tree(tree):

    def add_nodes_edges(tree, dot=None):
        # Create Digraph object
        if dot is None:
            dot = Digraph()
            dot.node(name=str(tree), label=str(tree.val))

        # Add nodes
        if tree.left:
            dot.node(name=str(tree.left), label=str(tree.left.val))
            dot.edge(str(tree), str(tree.left))
            dot = add_nodes_edges(tree.left, dot=dot)

        if tree.right:
            dot.node(name=str(tree.right), label=str(tree.right.val))
            dot.edge(str(tree), str(tree.right))
            dot = add_nodes_edges(tree.right, dot=dot)

        return dot

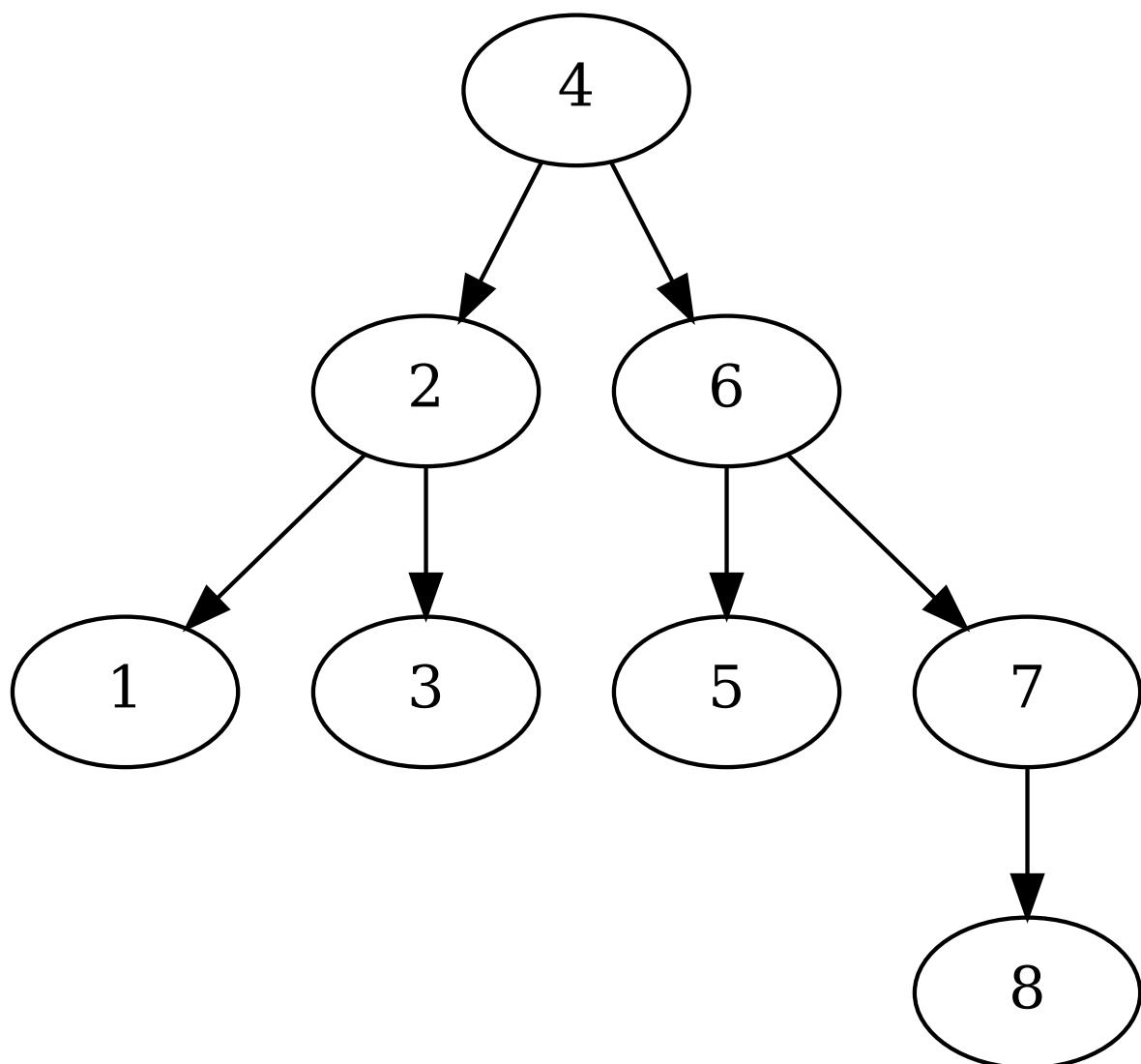
    # Add nodes recursively and create a list of edges
    dot = add_nodes_edges(tree)

    # Visualize the graph
    display(dot)

    return dot
```

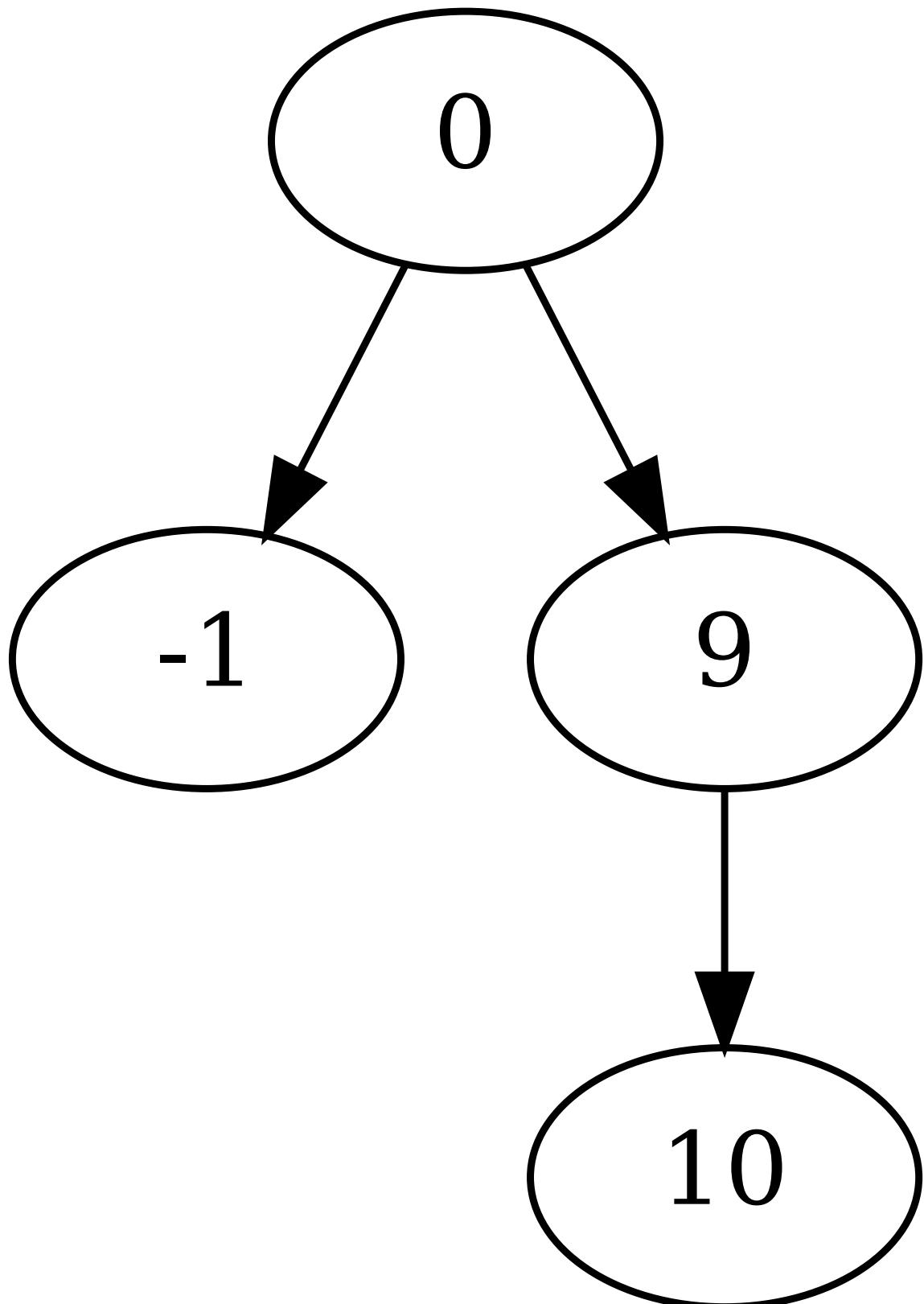
CELL 29

```
dot = visualize_tree(test1_tree)
```



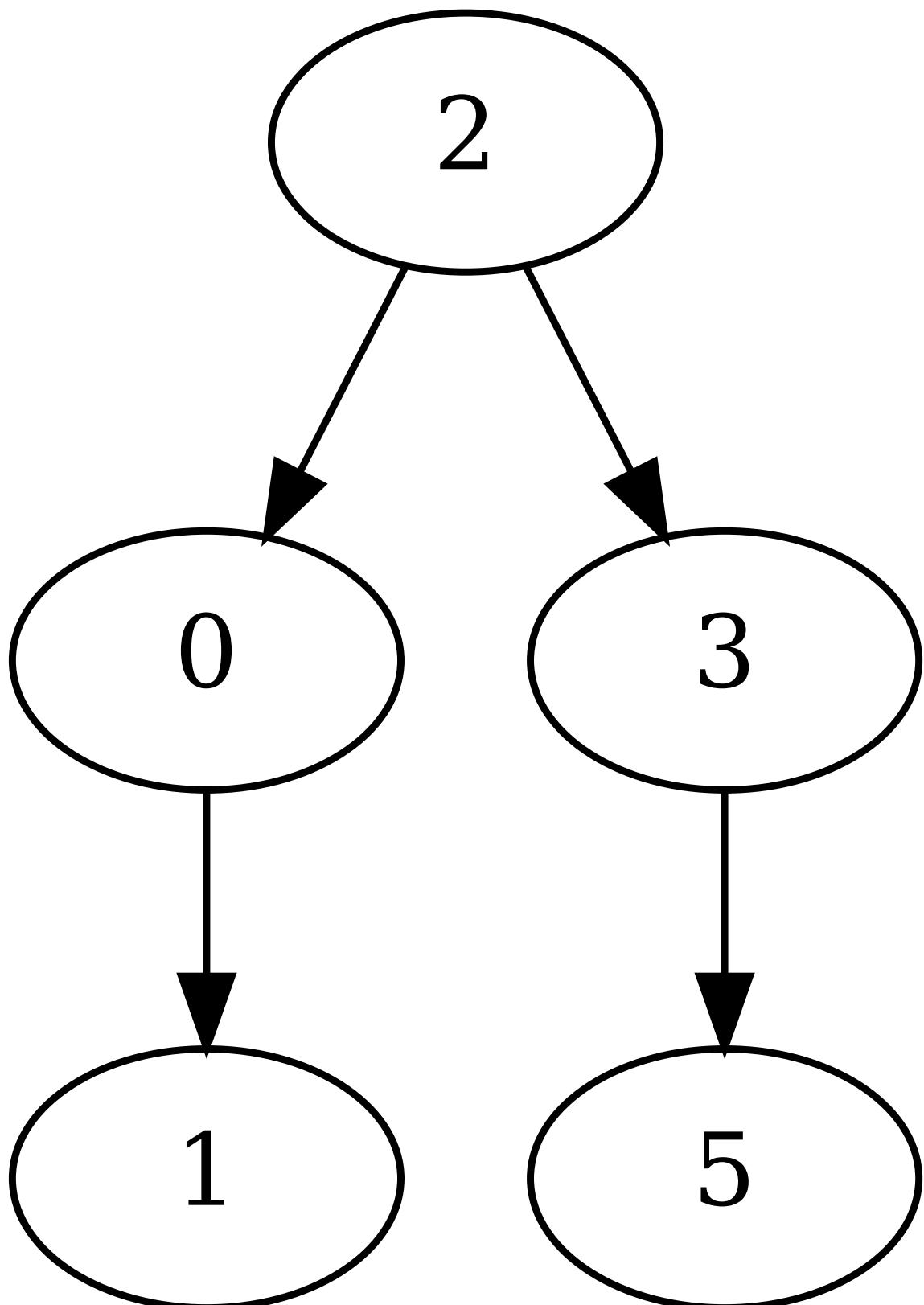
CELL 30

```
dot = visualize_tree(test2_tree)
```



CELL 31

```
dot = visualize_tree(test3_tree)
```



1.0.4 Section 3: Implementing Search, Insert and Delete Operations

Let us implement IDS operations on the BST we have built
Search operation should look like:

CELL 36

```
def search(nodes, val):  
  
    if val == nodes.val:  
        return True  
  
    if val < nodes.val:  
  
        if nodes.left == None:  
            return False  
  
        return search(nodes.left, val)  
  
    elif val >= nodes.val:  
  
        if nodes.right == None:  
            return False  
  
        return search(nodes.right, val)
```

CELL 37

```
search(test3_tree, 3)
```

True

CELL 38

```
search(test1_tree, 18)
```

```
False
```

Let us write insert function now:

CELL 40

```
def insert(nodes, val):

    # Empty Tree
    if nodes == None:
        nodes = Node(val)
        return

    # Value already exist on the node
    if nodes.val == val:
        return

    if val < nodes.val:

        if nodes.left == None:
            nodes.left = Node(val)
            return
        else:
            insert(nodes.left, val)
            return

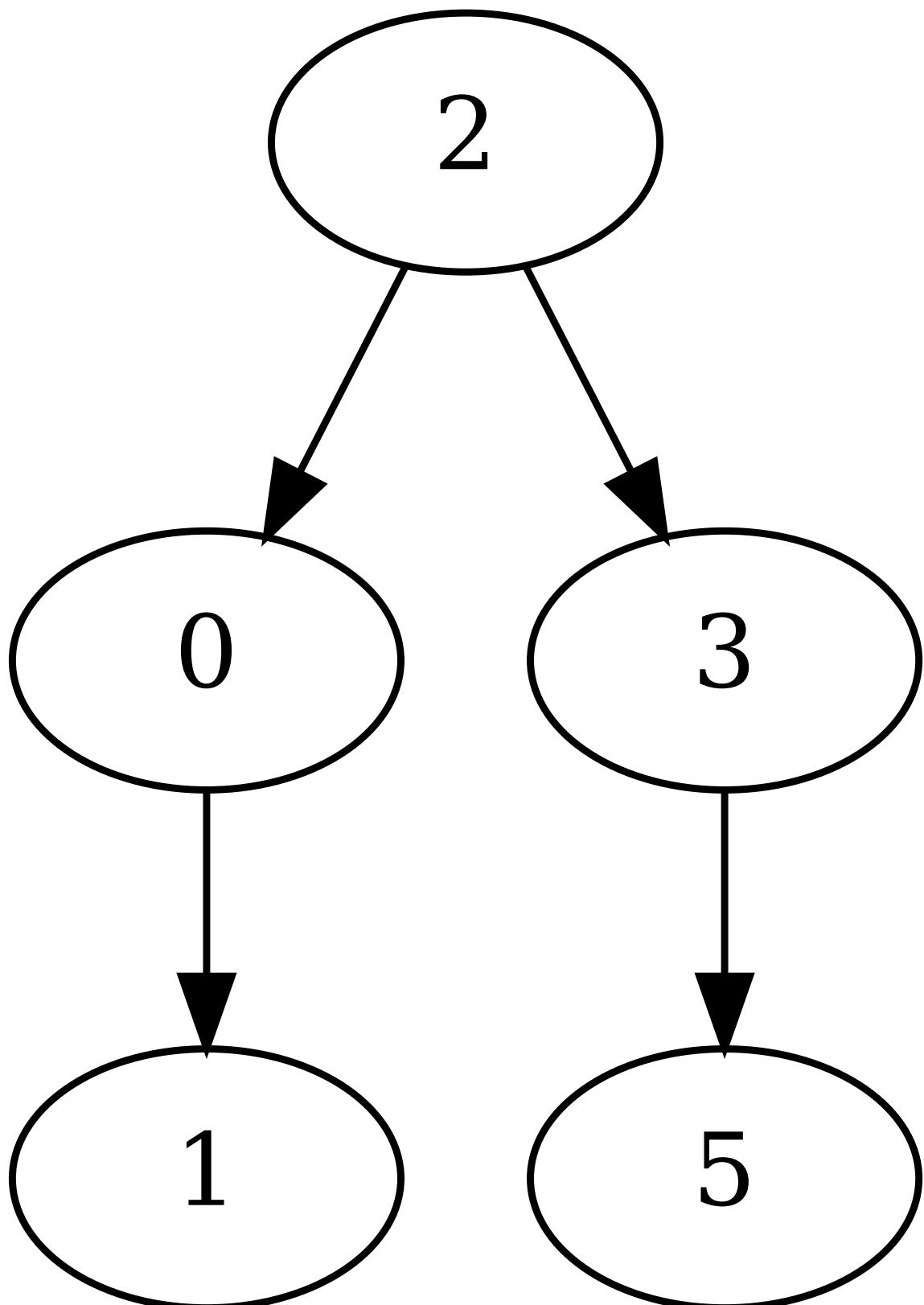
    elif val >= nodes.val:

        if nodes.right == None:
            nodes.right = Node(val)
            return
        else:
            insert(nodes.right, val)
            return
```

CELL 41

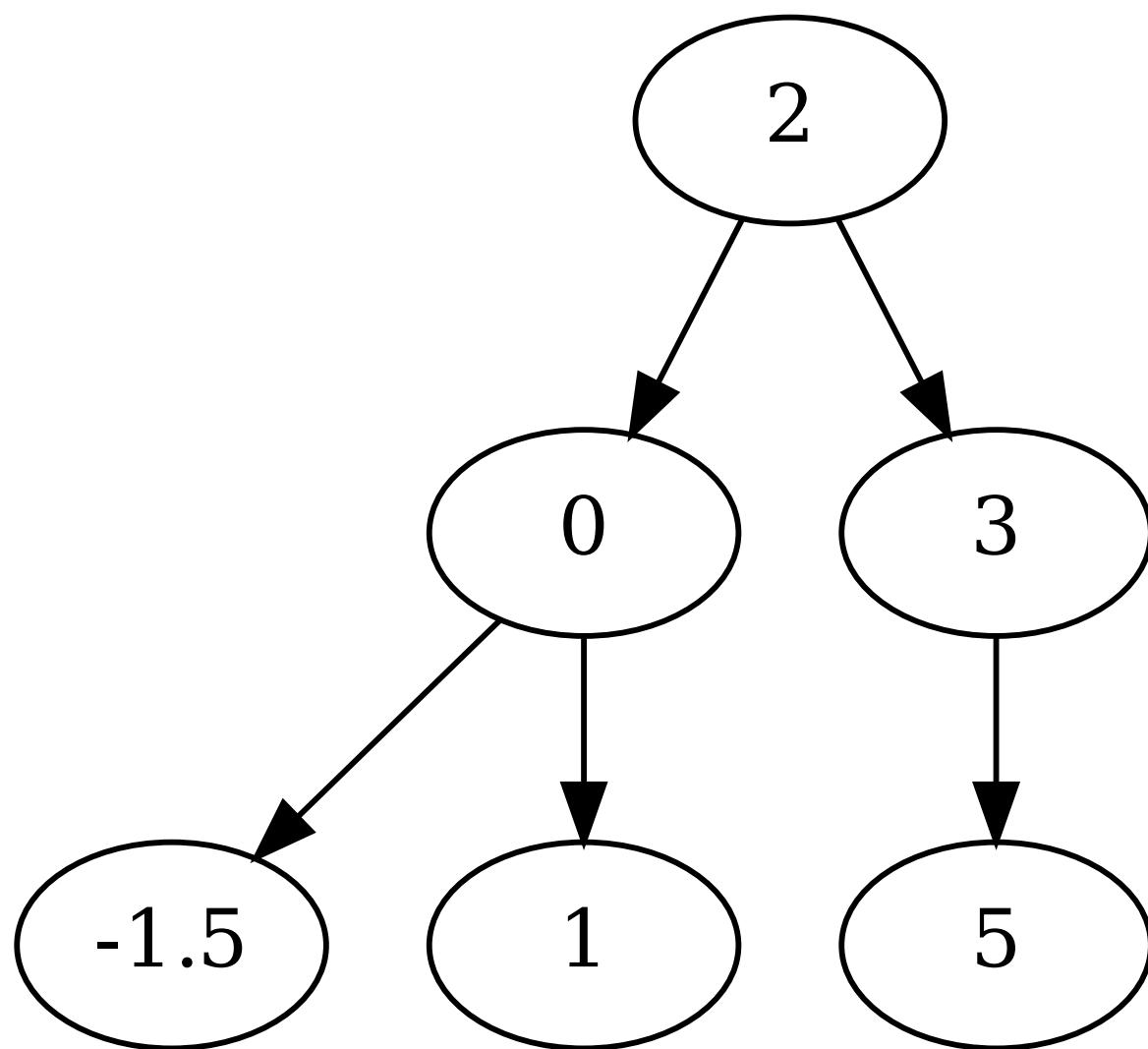
```
test3 = [0, 1, 2, 3, 3, 3, 5]
test3 = np.unique(test3)

test3_tree = buildBinaryTree(test3)
dot = visualize_tree(test3_tree)
```



CELL 42

```
insert(test3_tree, -1.5)
dot = visualize_tree(test3_tree)
```



Let us write delete operation. We will write another function minValueNode as well.

CELL 44

```
def minValueNode(node):
    current = node

    # loop down to find the leftmost leaf
    while(current.left is not None):
        current = current.left

    return current

def delete(nodes, val):

    if nodes == None:
        return nodes

    if val < nodes.val:

        #if nodes.left:
        nodes.left = delete(nodes.left, val)

    elif val > nodes.val:

        #if nodes.right:
        nodes.right = delete(nodes.right, val)

    else:

        # Node with only one child or no child

        if nodes.left is None:
            temp = nodes.right
            nodes = None
            return temp
        elif nodes.right is None:
            temp = nodes.left
            nodes = None
            return temp

        # Nodes with two children: Get the inorder successor
        temp = minValueNode(nodes.right)

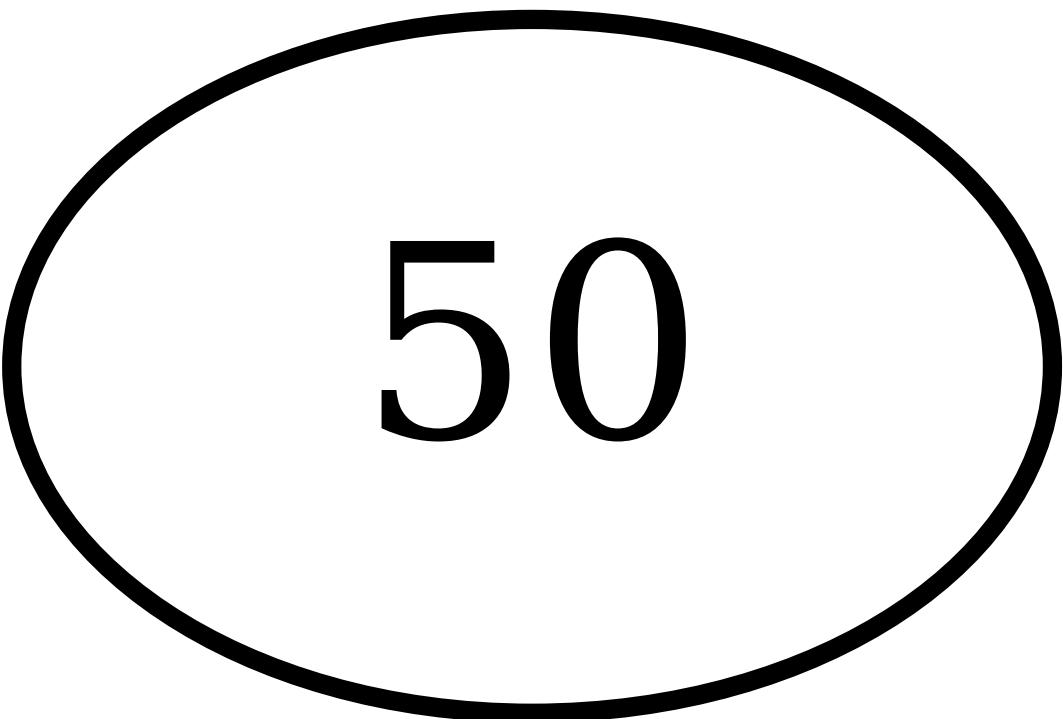
        nodes.val = temp.val

        nodes.right = delete(nodes.right, temp.val)

    return nodes
```

CELL 45

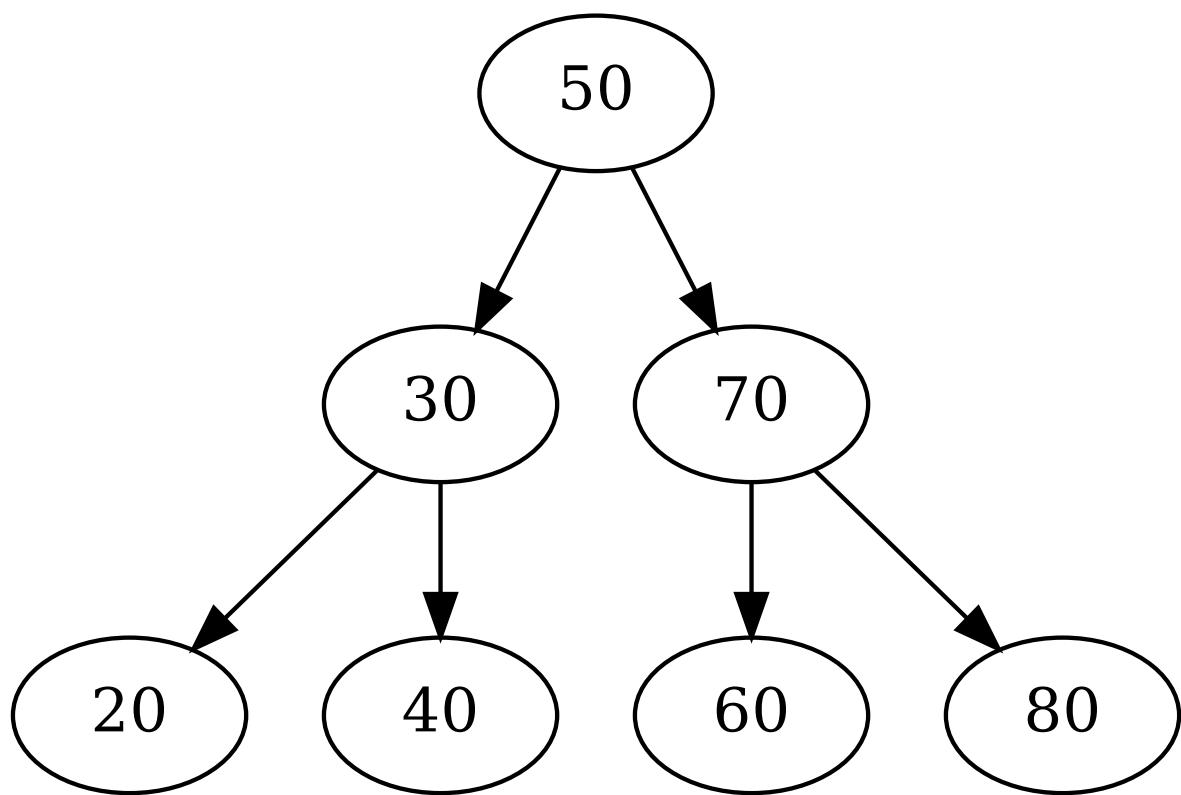
```
test3_tree = buildBinaryTree([50])
dot = visualize_tree(test3_tree)
```



50

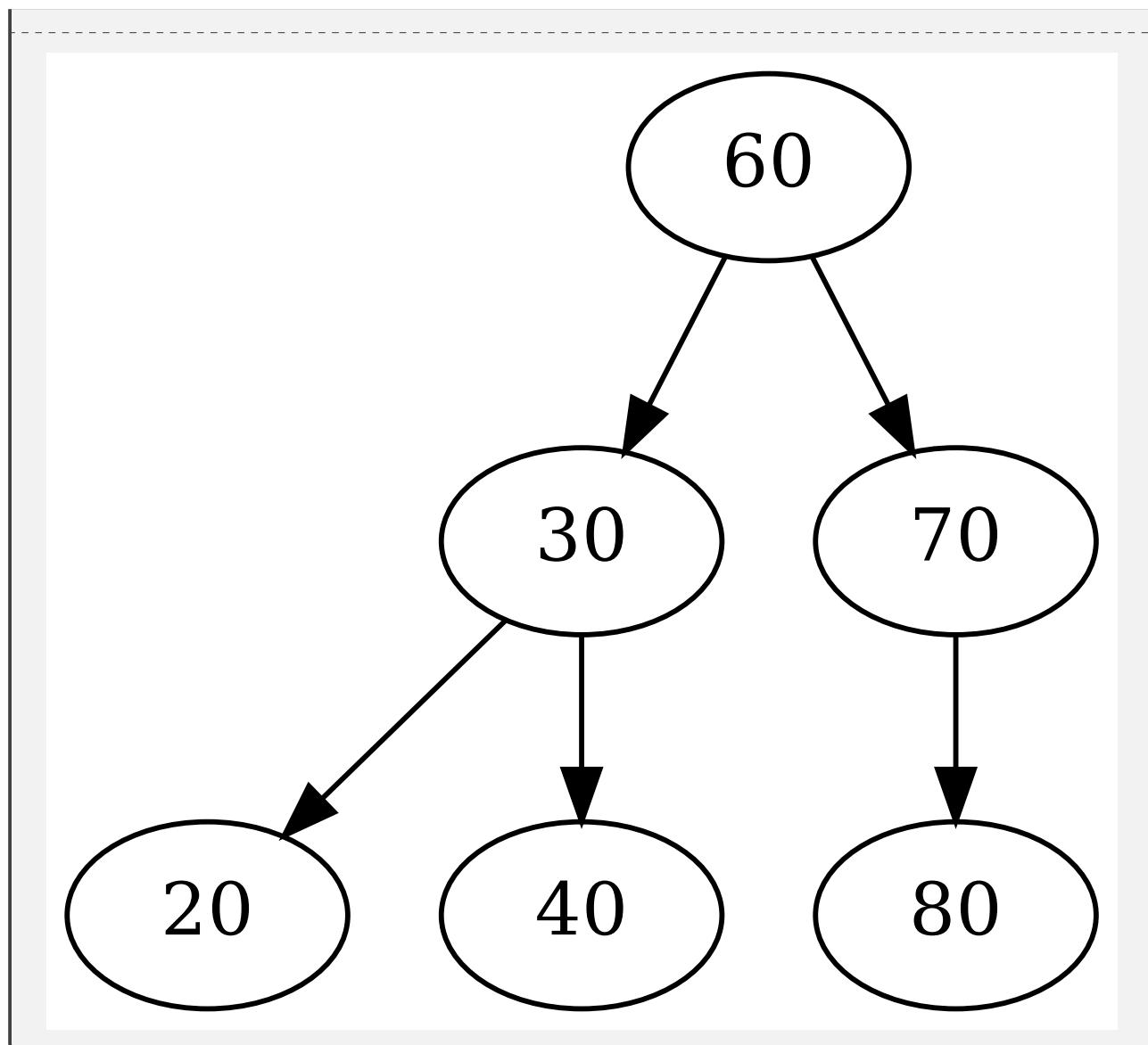
CELL 46

```
insert(test3_tree, 50)
insert(test3_tree, 30)
insert(test3_tree, 20)
insert(test3_tree, 40)
insert(test3_tree, 70)
insert(test3_tree, 60)
insert(test3_tree, 80)
dot = visualize_tree(test3_tree)
```



CELL 47

```
delete(test3_tree, 50)
dot = visualize_tree(test3_tree)
```



1.0.5 Section 4: Now that you have a good understanding of BST, write down code for activities in the onTrack Task sheet, in the following section

CELL 50

```
#Task 2
class BalancedNode:
    def __init__(self, value):
        self.val = value
        self.left = None
        self.right = None
        self.height = 0
        self.balance = 0

    def update_height(self):
        left_height = self.left.height if self.left else -1
        right_height = self.right.height if self.right else -1
        self.height = max(left_height, right_height) + 1
        self.balance = left_height - right_height

    def insert_balanced(self, val):
        if not self:
            return BalancedNode(val)

        if val < self.val:
            self.left = insert_balanced(self.left, val)
        else:
            self.right = insert_balanced(self.right, val)

        update_height(self)
        return self

    def is_balanced(self):
        if not self:
            return True
        return abs(self.balance) <= 1 and is_balanced(self.left) and is_balanced(self.right)

# Build and visualize balanced tree
balanced_tree = None
for val in [5, 3, 7, 2, 4, 6, 8]:
    balanced_tree = insert_balanced(balanced_tree, val)

def visualize_balanced_tree(tree):
    def add_nodes(tree, dot=None):
        if dot is None:
            dot = Digraph()
            label = f"{tree.val}\nH={tree.height}\nB={tree.balance}"
            dot.node(name=str(id(tree)), label=label)

        if tree.left:
            left_label = f"{tree.left.val}\nH={tree.left.height}\nB={tree.left.balance}"
            dot.node(name=str(id(tree.left)), label=left_label)
            dot.edge(str(id(tree)), str(id(tree.left)))
            add_nodes(tree.left, dot)

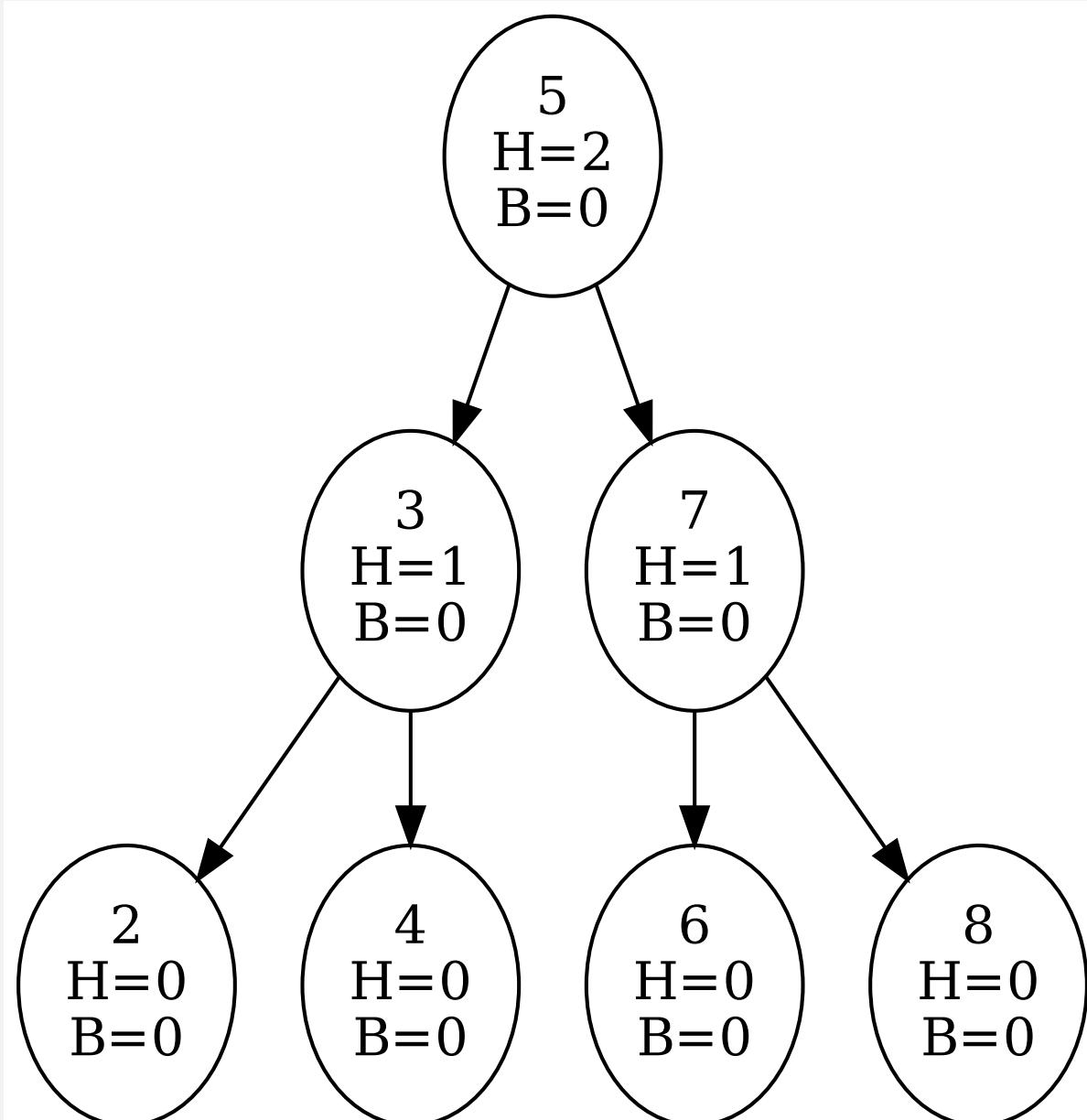
        if tree.right:
            right_label =
                f"{tree.right.val}\nH={tree.right.height}\nB={tree.right.balance}"
            dot.node(name=str(id(tree.right)), label=right_label)
            dot.edge(str(id(tree)), str(id(tree.right)))
            add_nodes(tree.right, dot)

        return dot

    return add_nodes(balanced_tree)
```

```
dot = add_nodes(tree)
display(dot)
return dot

visualize_balanced_tree(balanced_tree)
print("Is balanced:", is_balanced(balanced_tree))
```



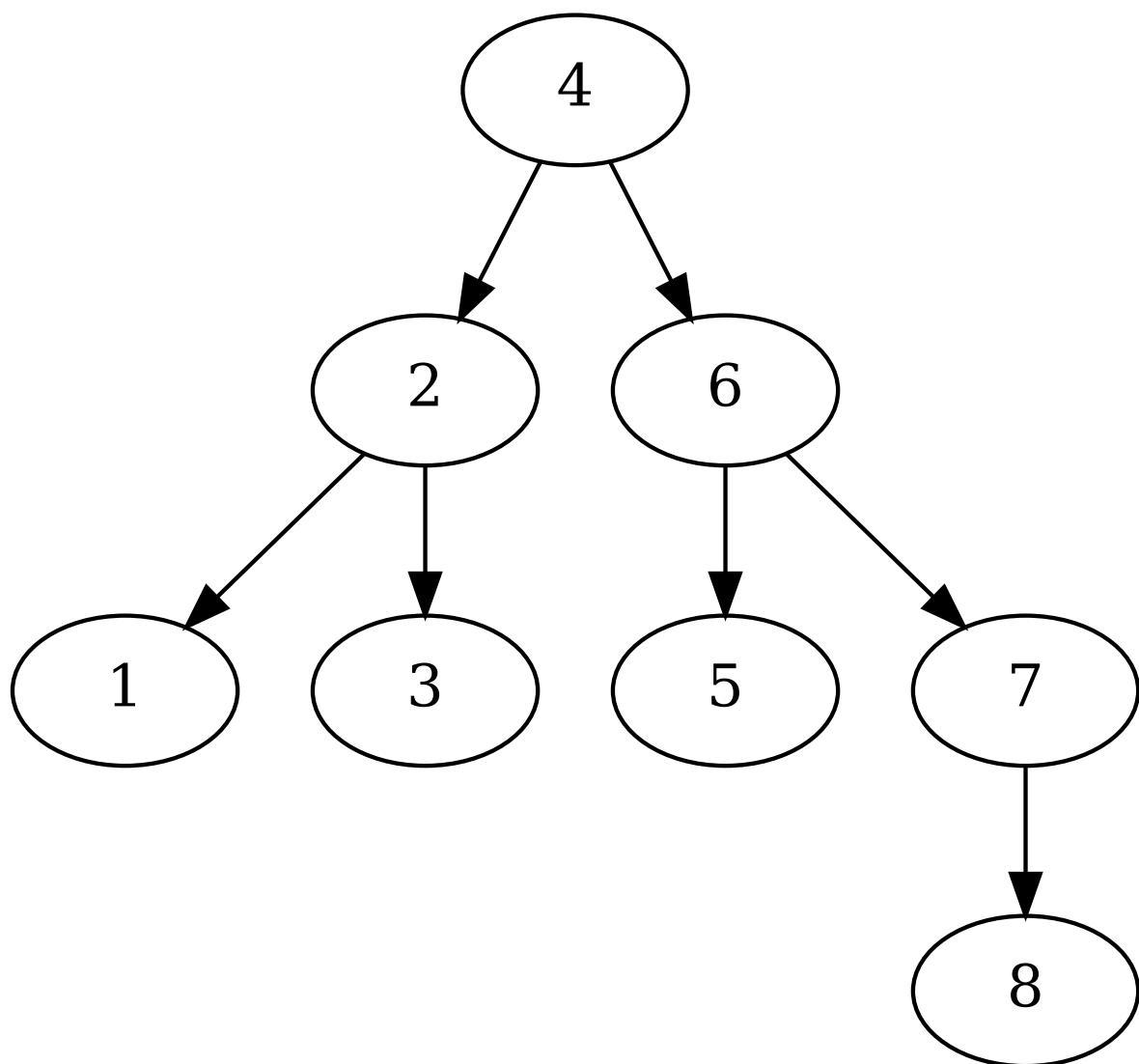
Is balanced: True

CELL 51

```
#Task 3

def find_common_ancestor(root, p, q):
    current = root
    while current:
        if p < current.val and q < current.val:
            current = current.left
        elif p > current.val and q > current.val:
            current = current.right
        else:
            return current
    return None

# Test and visualize
common_ancestor = find_common_ancestor(test1_tree, 2, 6)
dot = visualize_tree(test1_tree)
if common_ancestor:
    print("Common ancestor of 2 and 6:", common_ancestor.val)
```



Common ancestor of 2 and 6: 4

CELL 52

```
#Task 4

def left_rotate(x):
    "Perform left rotation at node x"
    y = x.right
    if y is None:
        return x # Cannot rotate if no right child

    # Perform rotation
    x.right = y.left
    y.left = x
    return y

def right_rotate(y):
    "Perform right rotation at node y"
    x = y.left
    if x is None:
        return y # Cannot rotate if no left child

    # Perform rotation
    y.left = x.right
    x.right = y
    return x

def left_right_rotate(z):
    "Left-Right (Double) Rotation at node z"
    if z.left is None:
        return z

    # First perform left rotation on left child
    z.left = left_rotate(z.left)

    # Then perform right rotation on z
    return right_rotate(z)

def right_left_rotate(z):
    "Right-Left (Double) Rotation at node z"
    if z.right is None:
        return z

    # First perform right rotation on right child
    z.right = right_rotate(z.right)

    # Then perform left rotation on z
    return left_rotate(z)

def build_rotation_tree():
    root = Node(15)

    # Left subtree
    root.left = Node(5)
    root.left.left = Node(3)
    root.left.right = Node(10)
    root.left.right.left = Node(8)

    # Right subtree
    root.right = Node(20)
    root.right.left = Node(17)
    root.right.right = Node(25)

    return root

# Create test tree for rotations
```

```
print("Original tree:")
rotation_tree = build_rotation_tree()
dot_original = visualize_tree(rotation_tree)

print("\nAfter left rotation at root:")
try:
    rotated = left_rotate(rotation_tree)
    dot_left = visualize_tree(rotated)
except Exception as e:
    print(f"Cannot perform left rotation: {str(e)}")

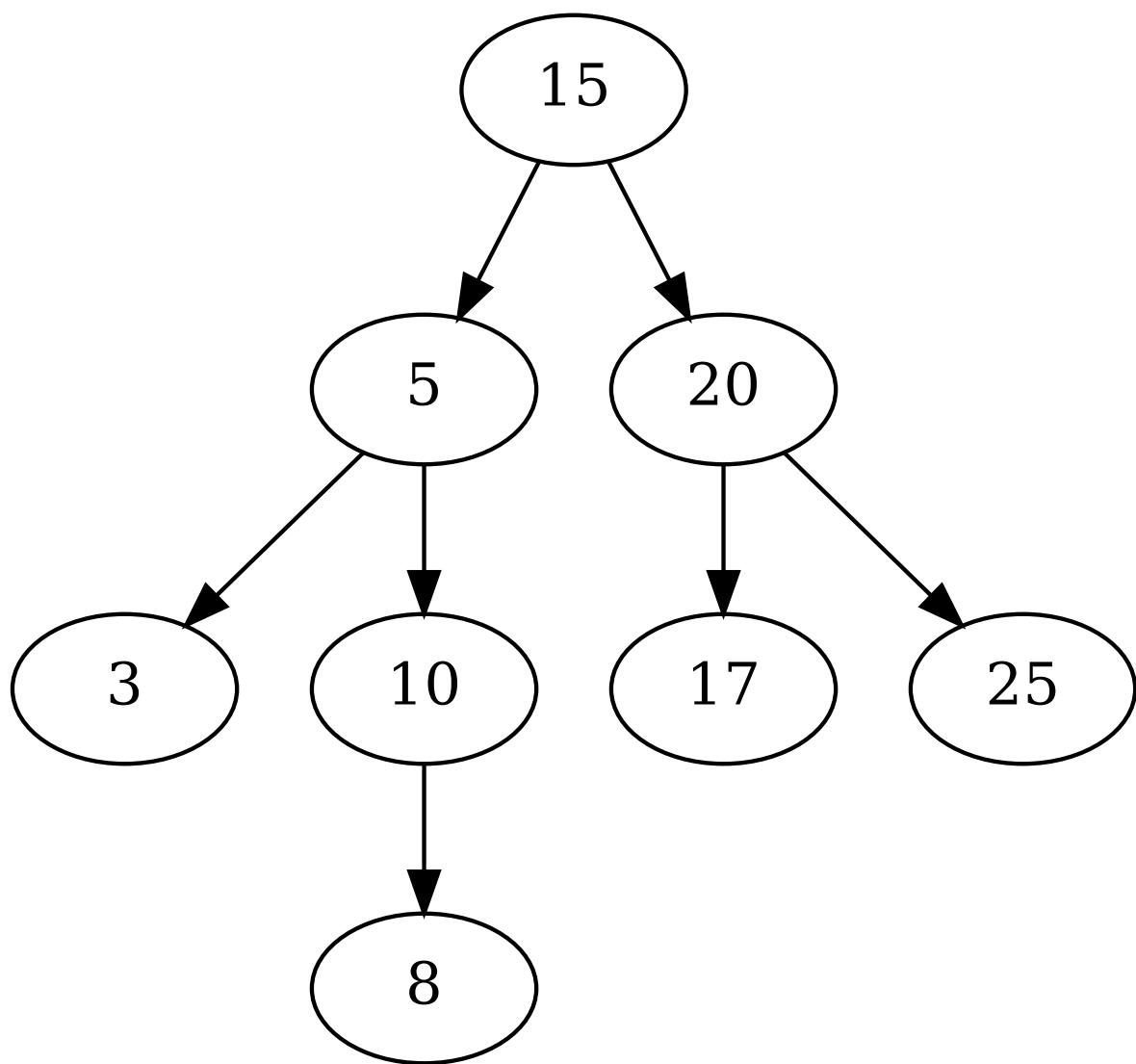
print("\nAfter right rotation at root:")
try:
    rotated = right_rotate(rotation_tree)
    dot_right = visualize_tree(rotated)
except Exception as e:
    print(f"Cannot perform right rotation: {str(e)}")

# Create a copy for double rotations
double_rot_tree = build_rotation_tree()

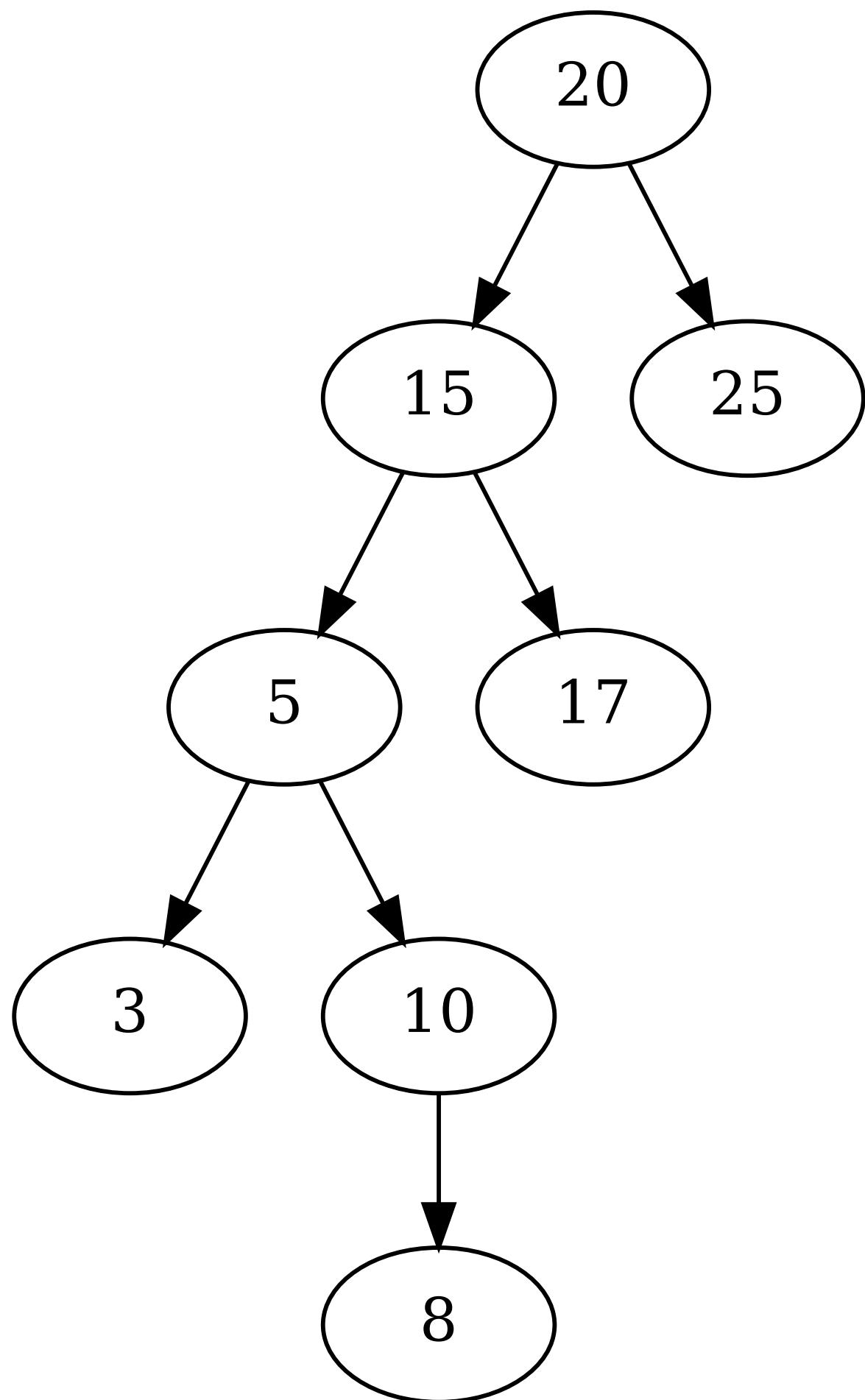
print("\nAfter left-right rotation at node 5:")
if double_rot_tree.left:
    # Perform left-right rotation on left child (node 5)
    double_rot_tree.left = left_right_rotate(double_rot_tree.left)
    dot_lr = visualize_tree(double_rot_tree)
else:
    print("No left child to rotate")

print("\nAfter right-left rotation at node 20:")
if double_rot_tree.right:
    # Perform right-left rotation on right child (node 20)
    double_rot_tree.right = right_left_rotate(double_rot_tree.right)
    dot_rl = visualize_tree(double_rot_tree)
else:
    print("No right child to rotate")
```

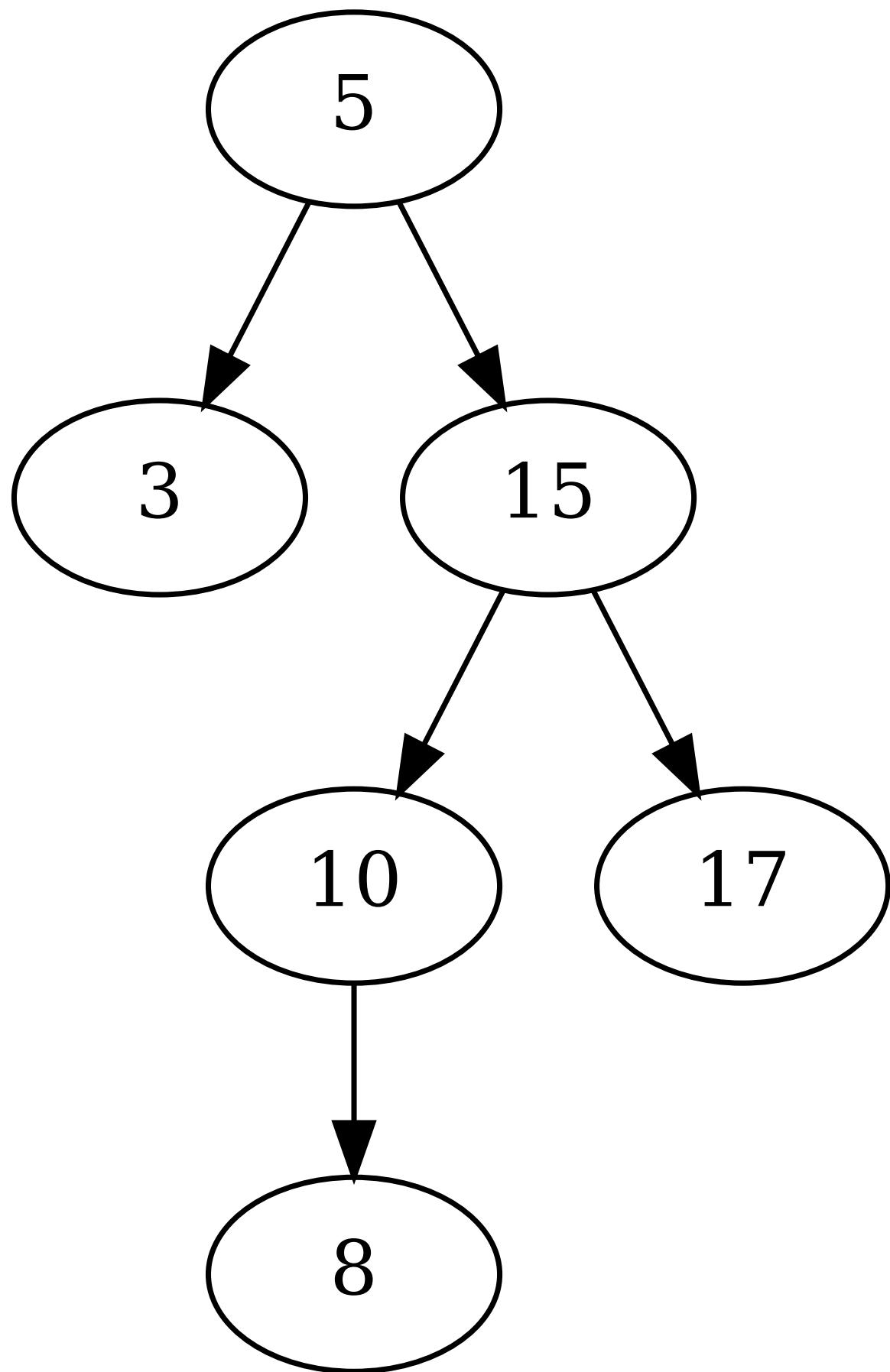
Original tree:



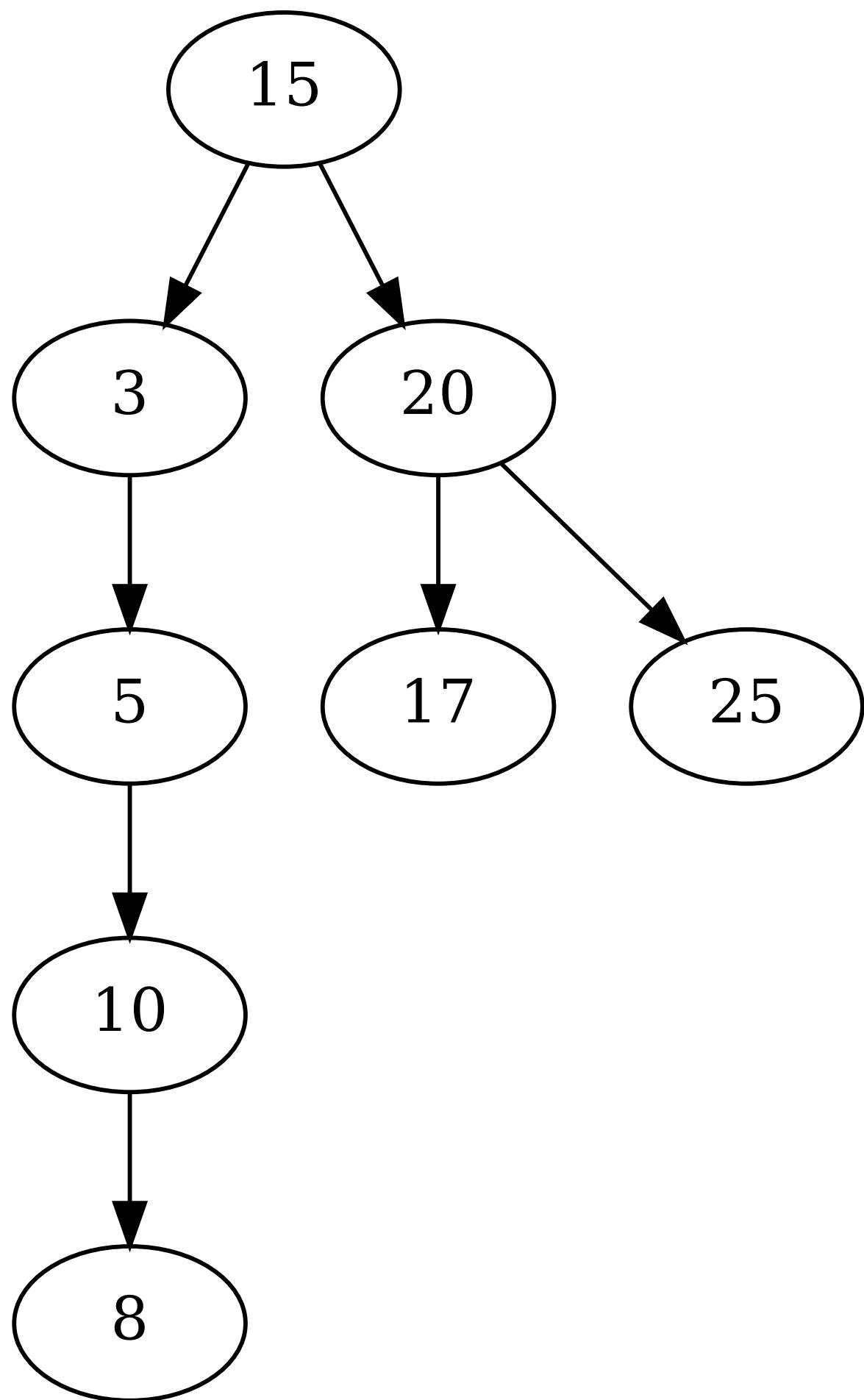
After left rotation at root:



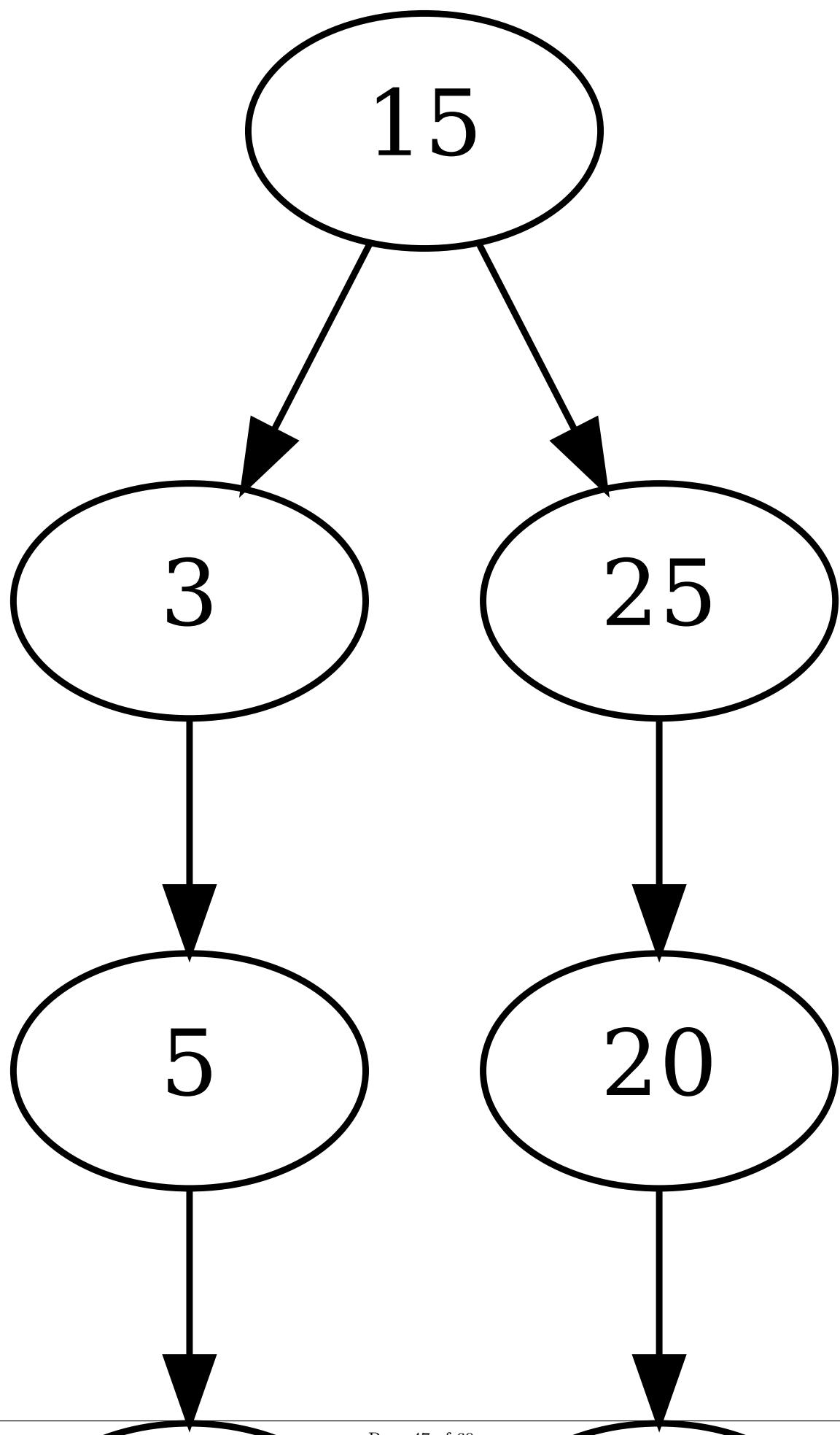
After right rotation at root:



After left-right rotation at node 5:



```
After right-left rotation at node 20:
```



CELL 53

```
#Task 5

RED = True
BLACK = False

class RBNode:
    def __init__(self, val, color=RED):
        self.val = val
        self.color = color
        self.left = None
        self.right = None
        self.parent = None

class RedBlackTree:
    def __init__(self):
        self.nil = RBNode(0, BLACK)
        self.nil.left = self.nil
        self.nil.right = self.nil
        self.nil.parent = self.nil
        self.root = self.nil

    def _left_rotate(self, x):
        "Left rotation at node x"
        y = x.right
        x.right = y.left

        if y.left != self.nil:
            y.left.parent = x

        y.parent = x.parent
        if x.parent == self.nil:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y

        y.left = x
        x.parent = y
        return y

    def _right_rotate(self, y):
        "Right rotation at node y"
        x = y.left
        y.left = x.right

        if x.right != self.nil:
            x.right.parent = y

        x.parent = y.parent
        if y.parent == self.nil:
            self.root = x
        elif y == y.parent.right:
            y.parent.right = x
        else:
            y.parent.left = x

        x.right = y
        y.parent = x
        return x

    def insert(self, val):
        "Insert value into tree"
```

```
new_node = RBNode(val)
new_node.left = self.nil
new_node.right = self.nil
new_node.parent = self.nil

current = self.root
parent = self.nil
while current != self.nil:
    parent = current
    if val < current.val:
        current = current.left
    else:
        current = current.right

    new_node.parent = parent
    if parent == self.nil:
        self.root = new_node
    elif val < parent.val:
        parent.left = new_node
    else:
        parent.right = new_node

    self._insert_fixup(new_node)
    return new_node

def _insert_fixup(self, node):
    "Fix red-black tree properties after insertion"
    while node.parent.color == RED:
        if node.parent == node.parent.parent.left:
            uncle = node.parent.parent.right
            if uncle.color == RED:
                # Case 1: Uncle is red
                node.parent.color = BLACK
                uncle.color = BLACK
                node.parent.parent.color = RED
                node = node.parent.parent
            else:
                if node == node.parent.right:
                    # Case 2: Uncle is black, node is right child
                    node = node.parent
                    self._left_rotate(node)
                # Case 3: Uncle is black, node is left child
                node.parent.color = BLACK
                node.parent.parent.color = RED
                self._right_rotate(node.parent.parent)
        else:
            # Mirror cases
            uncle = node.parent.parent.left
            if uncle.color == RED:
                node.parent.color = BLACK
                uncle.color = BLACK
                node.parent.parent.color = RED
                node = node.parent.parent
            else:
                if node == node.parent.left:
                    node = node.parent
                    self._right_rotate(node)
                node.parent.color = BLACK
                node.parent.parent.color = RED
                self._left_rotate(node.parent.parent)

        if node == self.root:
            break

    self.root.color = BLACK
```

```
def _transplant(self, u, v):
    "Replace subtree rooted at u with subtree rooted at v"
    if u.parent == self.nil:
        self.root = v
    elif u == u.parent.left:
        u.parent.left = v
    else:
        u.parent.right = v
    v.parent = u.parent

def _minimum(self, node):
    "Find minimum node in subtree"
    while node.left != self.nil:
        node = node.left
    return node

def _find_node(self, val):
    "Find node with given value"
    current = self.root
    while current != self.nil:
        if val == current.val:
            return current
        elif val < current.val:
            current = current.left
        else:
            current = current.right
    return self.nil

def delete(self, val):
    "Delete value from tree"
    node = self._find_node(val)
    if node == self.nil:
        return

    y = node
    y_original_color = y.color
    if node.left == self.nil:
        x = node.right
        self._transplant(node, node.right)
    elif node.right == self.nil:
        x = node.left
        self._transplant(node, node.left)
    else:
        y = self._minimum(node.right)
        y_original_color = y.color
        x = y.right
        if y.parent == node:
            x.parent = y
        else:
            self._transplant(y, y.right)
            y.right = node.right
            y.right.parent = y

    self._transplant(node, y)
    y.left = node.left
    y.left.parent = y
    y.color = node.color

    if y_original_color == BLACK:
        self._delete_fixup(x)

def _delete_fixup(self, x):
    "Fix red-black tree properties after deletion"
    while x != self.root and x.color == BLACK:
        if x == x.parent.left:
            w = x.parent.right
```

```
if w.color == RED:
    # Case 1: Sibling is red
    w.color = BLACK
    x.parent.color = RED
    self._left_rotate(x.parent)
    w = x.parent.right

if w.left.color == BLACK and w.right.color == BLACK:
    # Case 2: Both siblings children are black
    w.color = RED
    x = x.parent
else:
    if w.right.color == BLACK:
        # Case 3: Sibling's right child is black
        w.left.color = BLACK
        w.color = RED
        self._right_rotate(w)
        w = x.parent.right

    # Case 4: Sibling's right child is red
    w.color = x.parent.color
    x.parent.color = BLACK
    w.right.color = BLACK
    self._left_rotate(x.parent)
    x = self.root

else:
    # Mirror cases
    w = x.parent.left
    if w.color == RED:
        w.color = BLACK
        x.parent.color = RED
        self._right_rotate(x.parent)
        w = x.parent.left

    if w.right.color == BLACK and w.left.color == BLACK:
        w.color = RED
        x = x.parent
    else:
        if w.left.color == BLACK:
            w.right.color = BLACK
            w.color = RED
            self._left_rotate(w)
            w = x.parent.left

            w.color = x.parent.color
            x.parent.color = BLACK
            w.left.color = BLACK
            self._right_rotate(x.parent)
            x = self.root

    x.color = BLACK

def visualize(self):
    "Visualize tree using Graphviz"
    dot = Digraph()
    self._add_nodes(self.root, dot)
    display(dot)
    return dot

def _add_nodes(self, node, dot):
    "Recursively add nodes to Graphviz diagram"
    if node == self.nil:
        return

    node_id = str(id(node))
    color = "red" if node.color == RED else "black"
```

```
font_color = "white" if node.color == BLACK else "black"
dot.node(node_id, label=str(node.val), style="filled", fillcolor=color,
fontcolor=font_color)

if node.left != self.nil:
    left_id = str(id(node.left))
    self._add_nodes(node.left, dot)
    dot.edge(node_id, left_id, label="L")

if node.right != self.nil:
    right_id = str(id(node.right))
    self._add_nodes(node.right, dot)
    dot.edge(node_id, right_id, label="R")

def print_step(self, action, val=None):
    "Print and visualize current step"
    if val is not None:
        print(f"\n{action} {val}:")
    else:
        print(f"\n{action}:")
    self.visualize()

# Create and visualize Red-Black Tree
rbt = RedBlackTree()

# Initial state
rbt.print_step("Initial state")

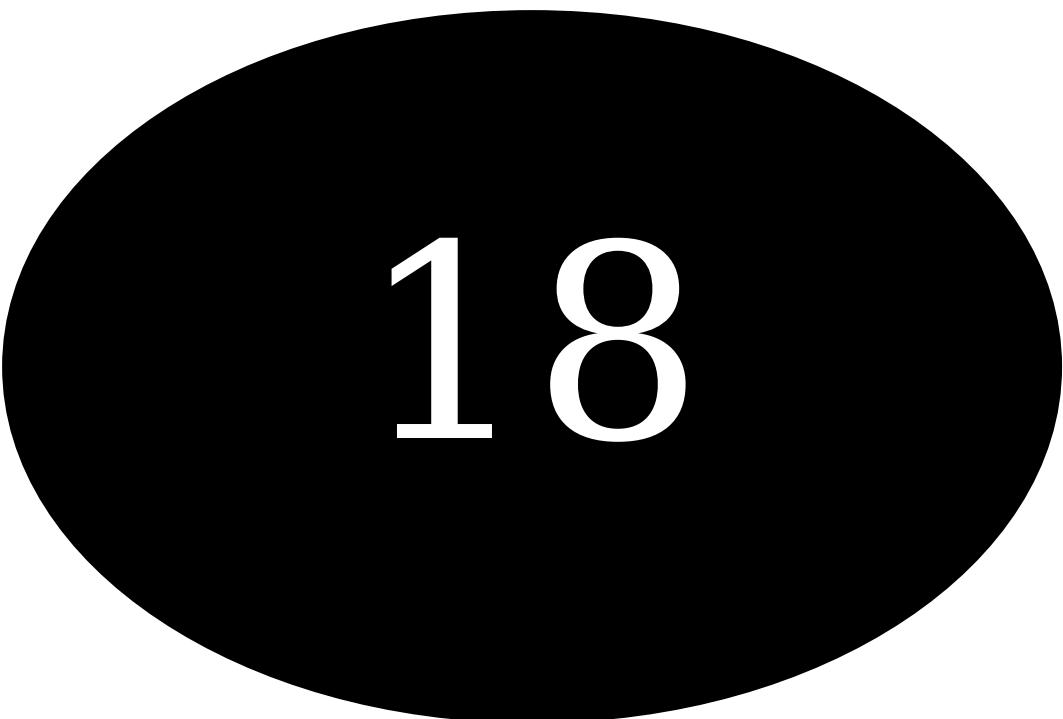
# Insertion sequence
values = [18, 2, 1, 12, 15, 9, 10, 13, 6, 7, 4, 29]
for val in values:
    rbt.insert(val)
    rbt.print_step("Inserting", val)

# Deletion sequence
deletions = [2, 29, 18]
for val in deletions:
    rbt.delete(val)
    rbt.print_step("Deleting", val)
```

Initial state:

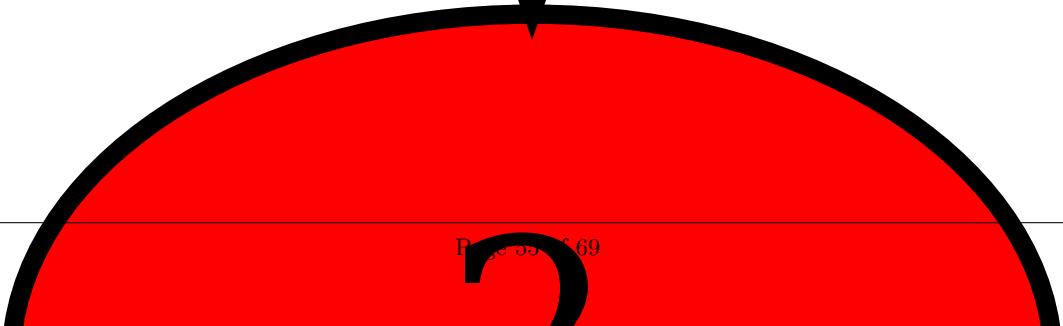
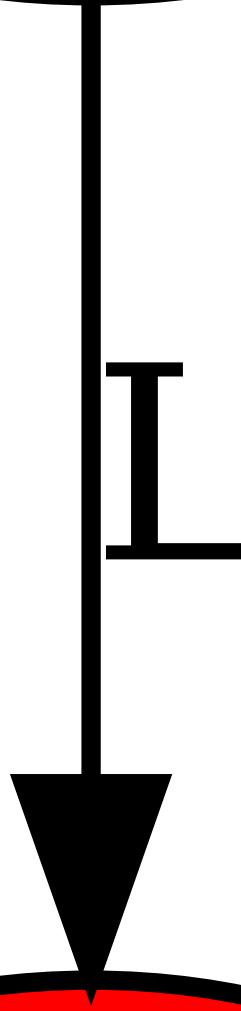
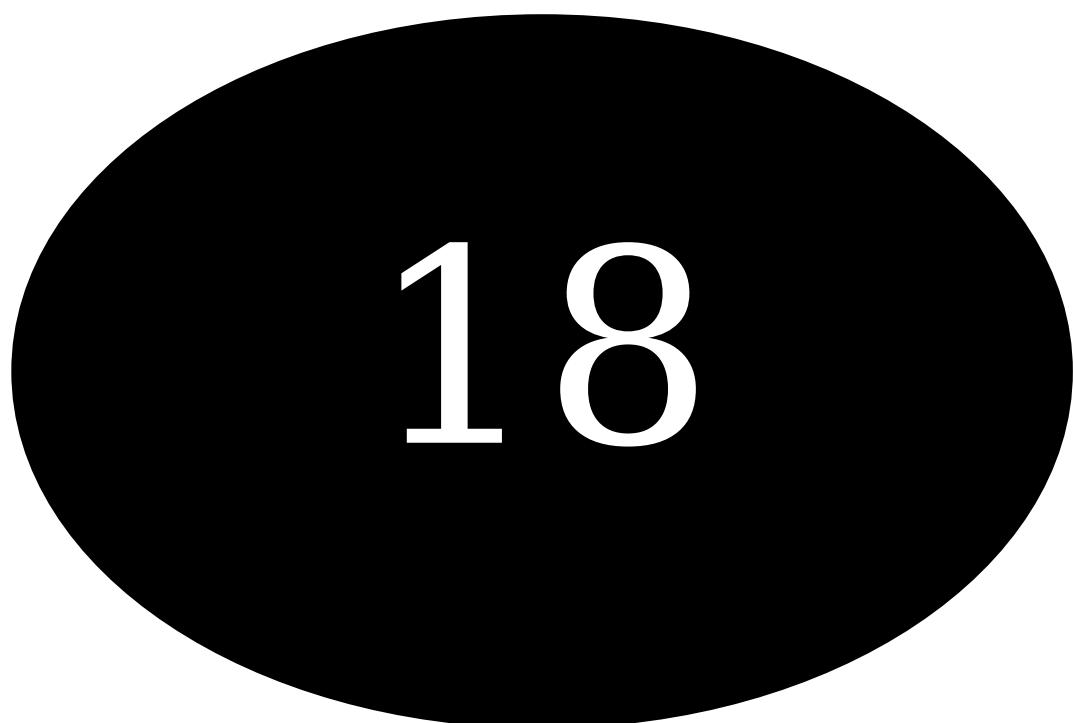


Inserting 18:

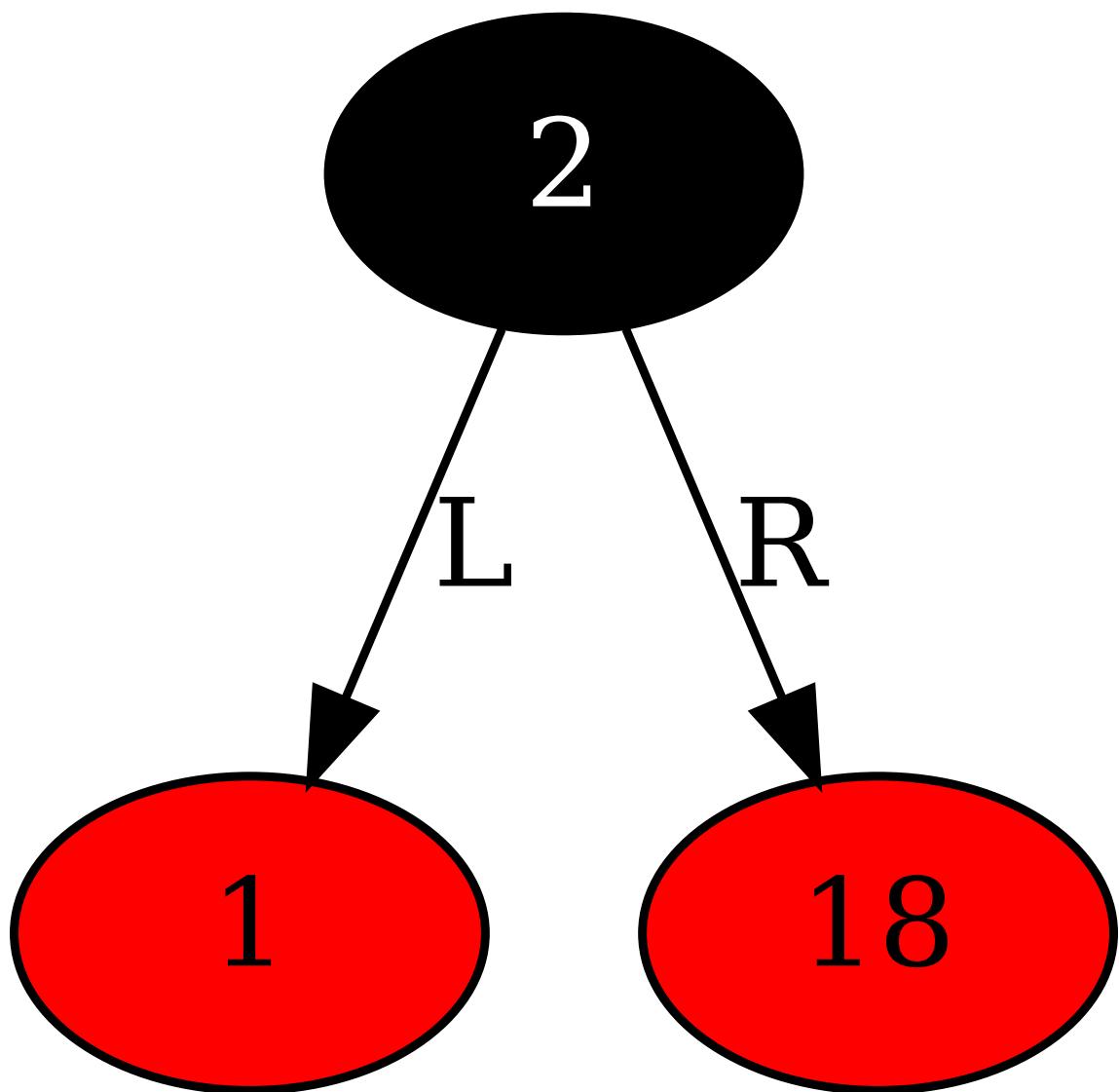


18

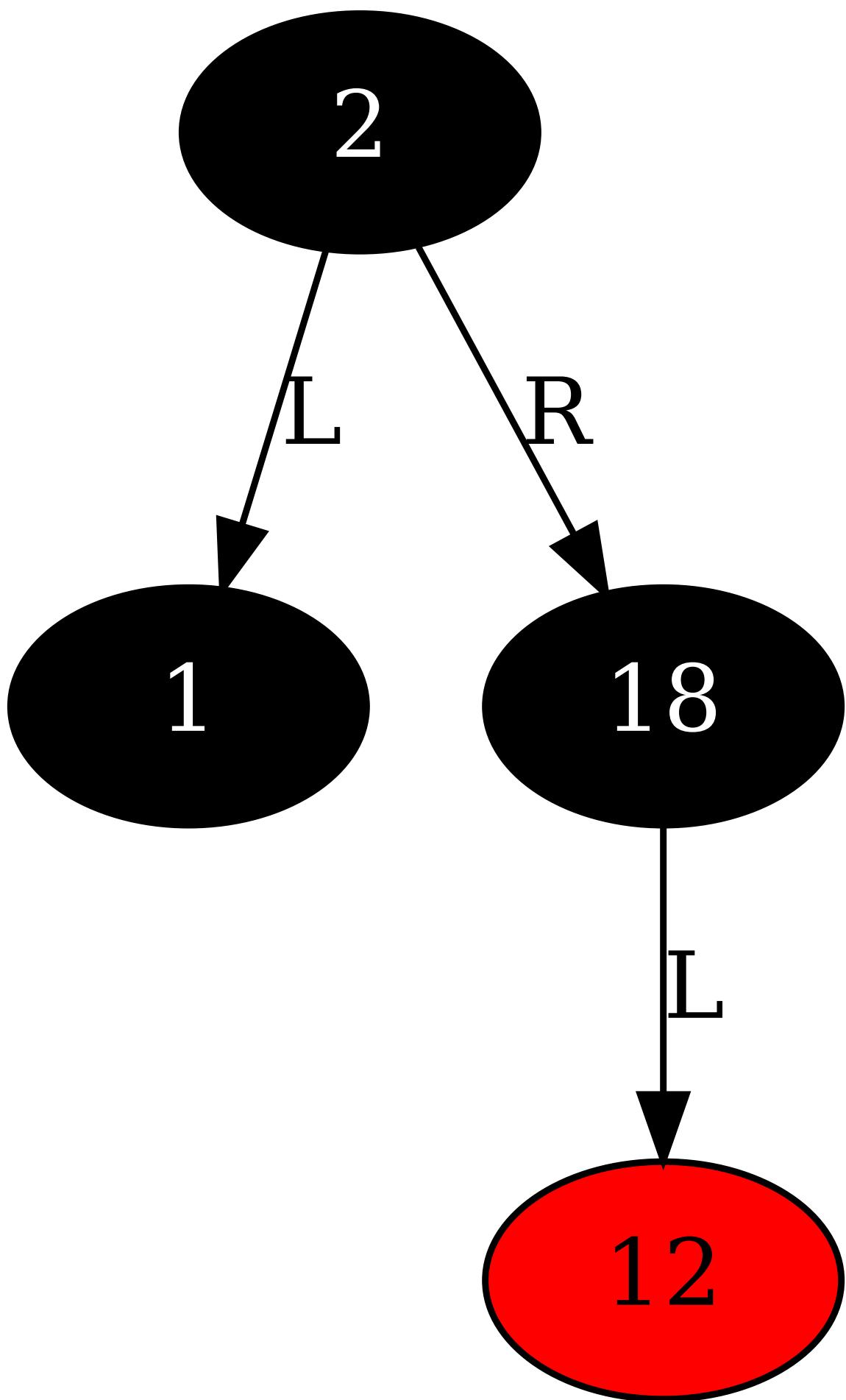
Inserting 2:



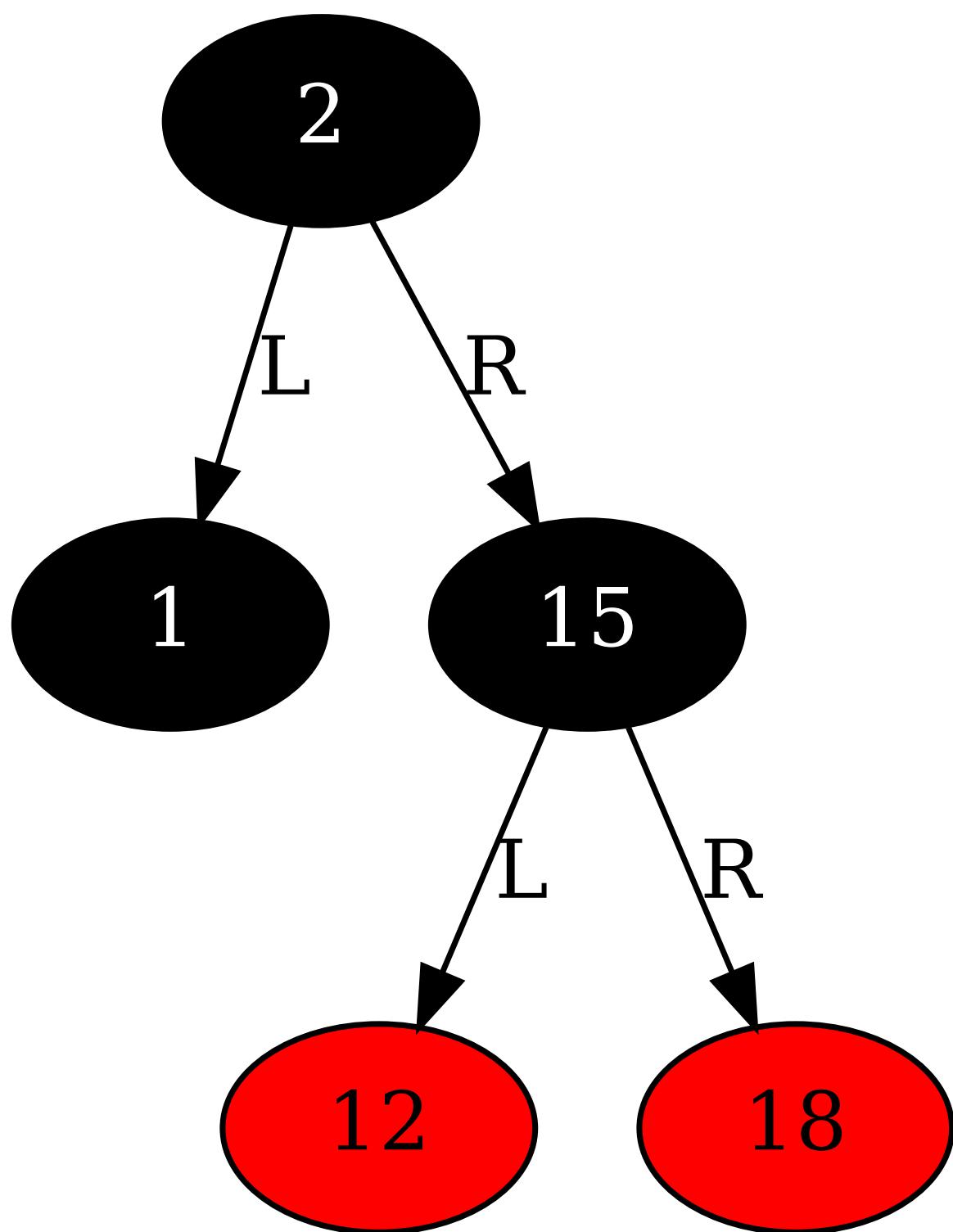
Inserting 1:



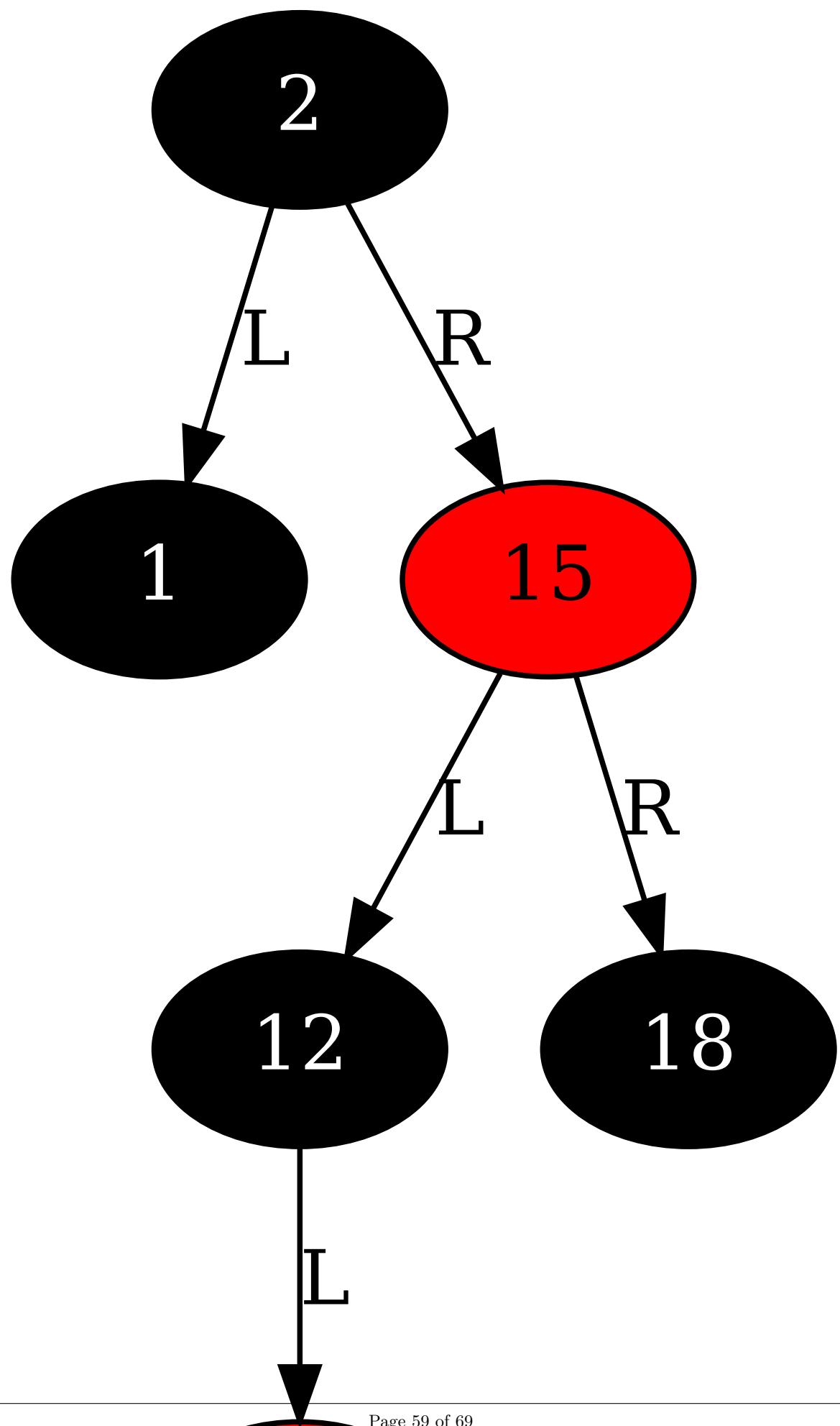
Inserting 12:



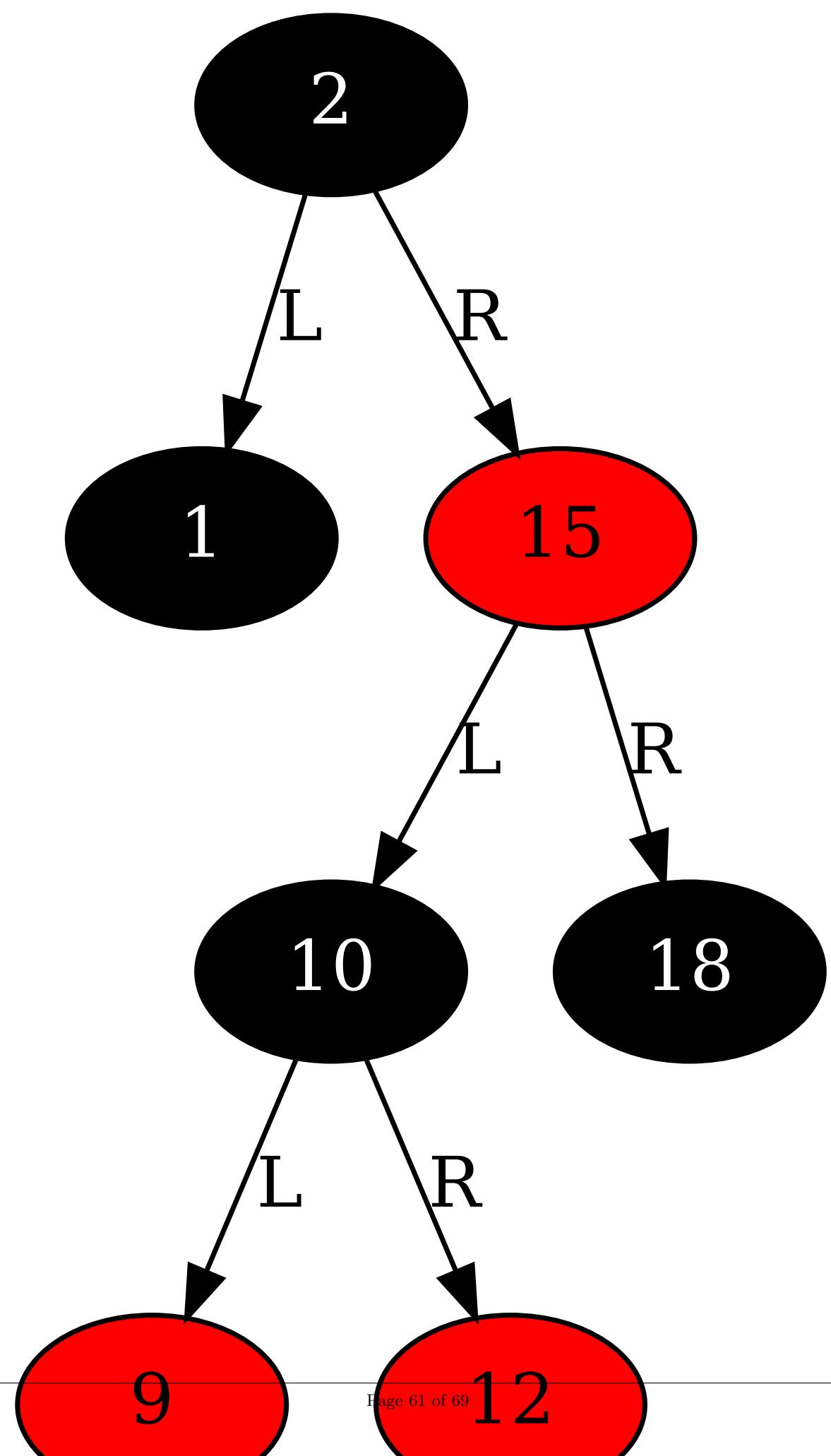
Inserting 15:



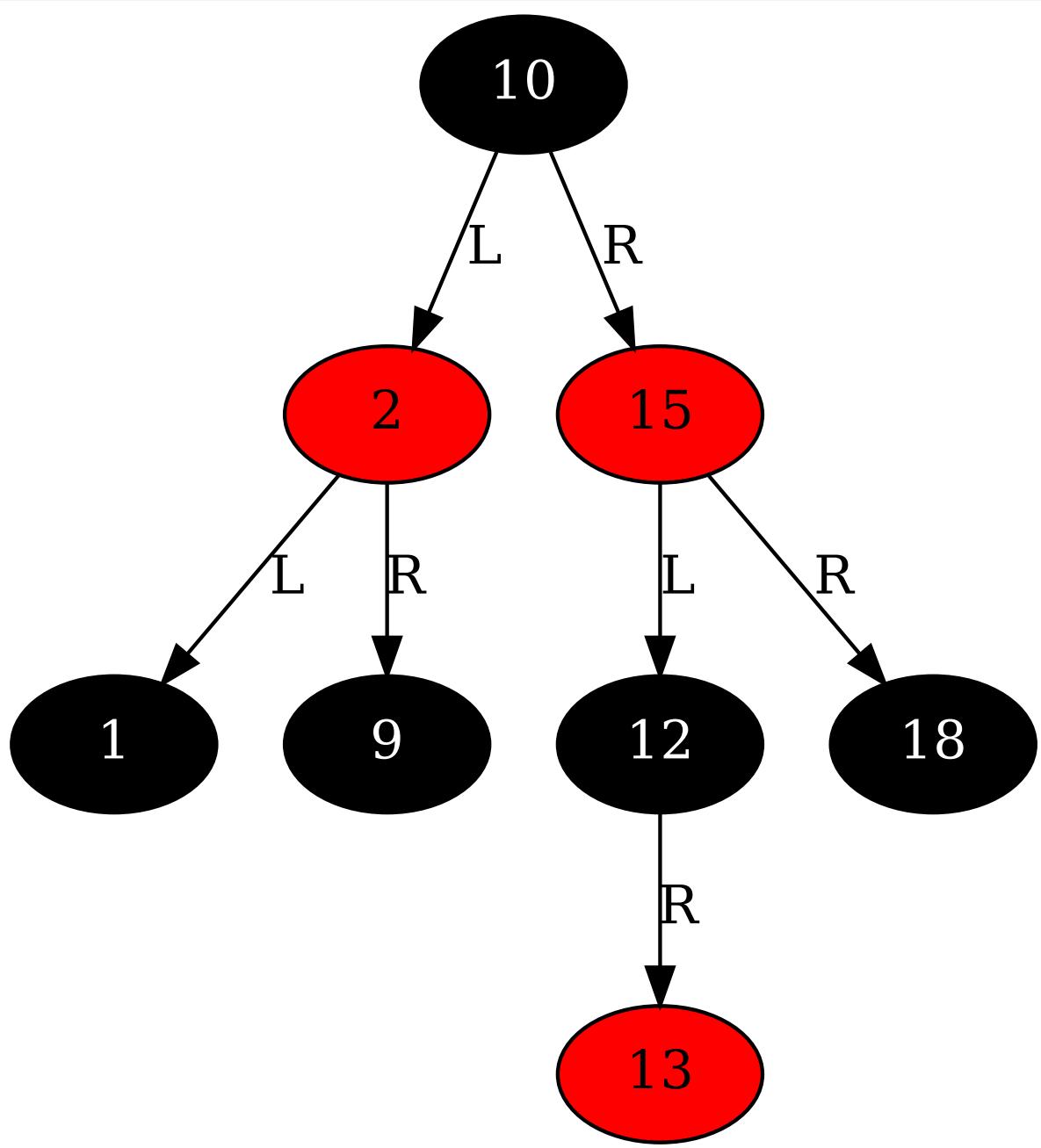
Inserting 9:



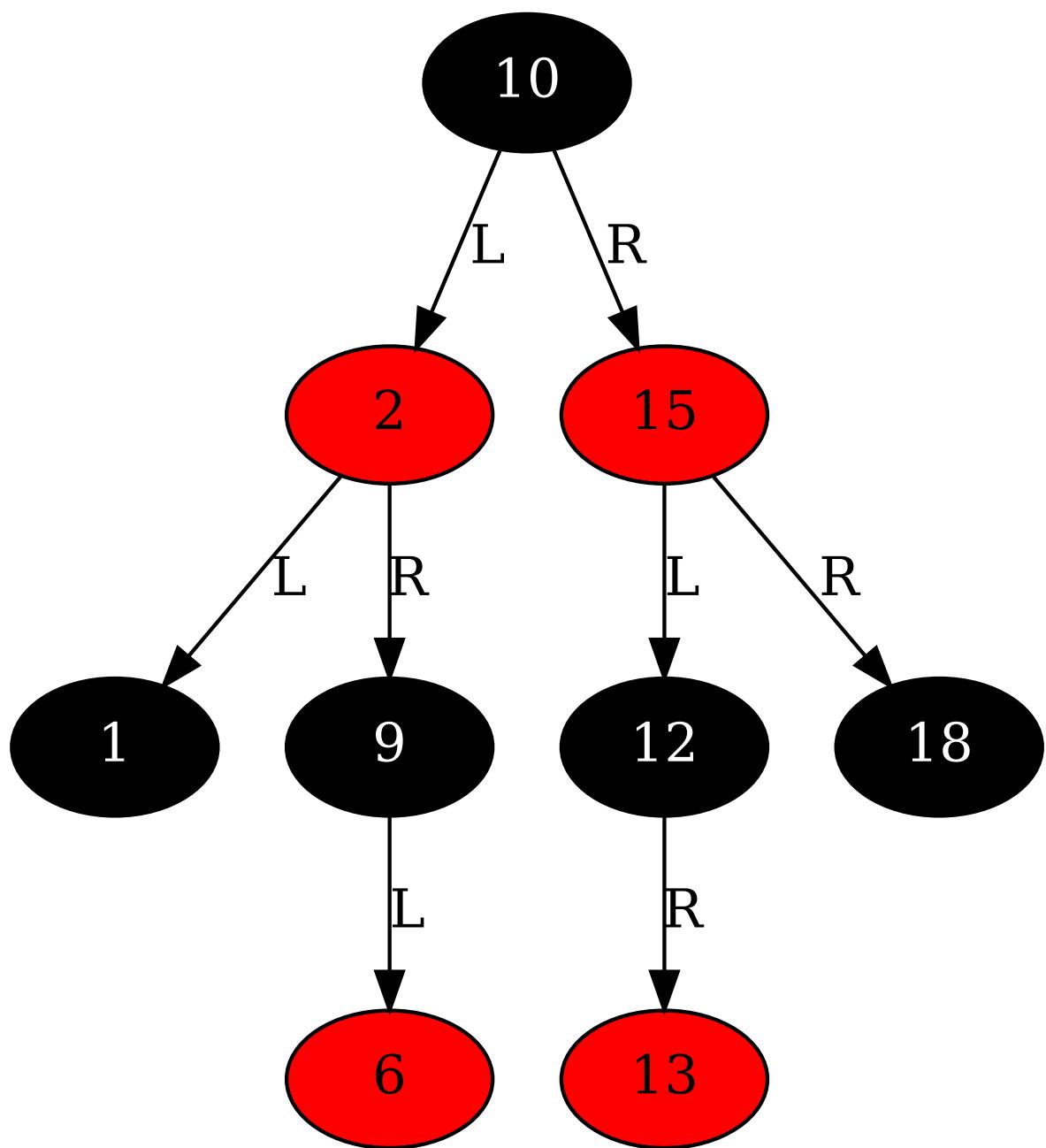
Inserting 10:



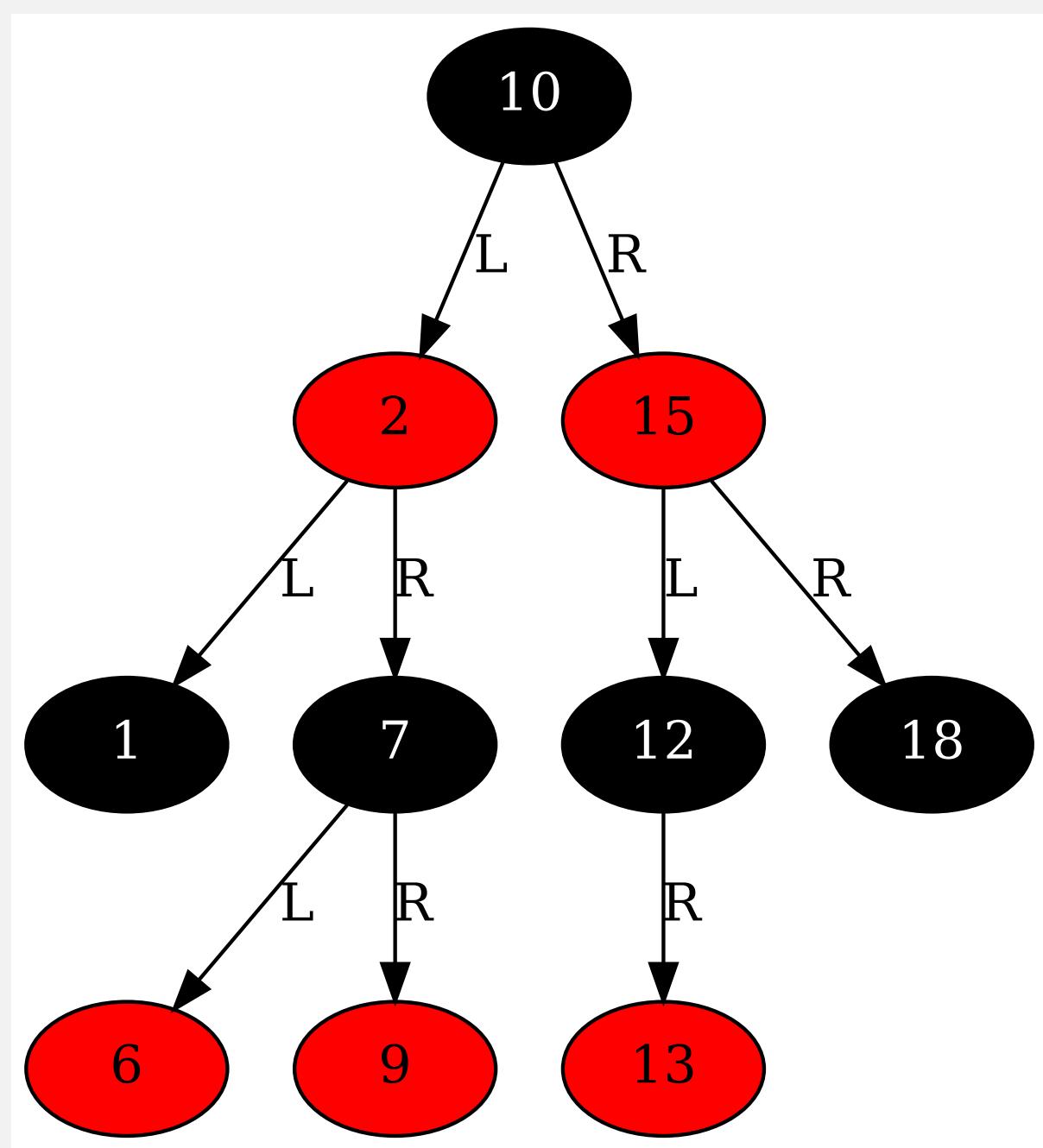
Inserting 13:



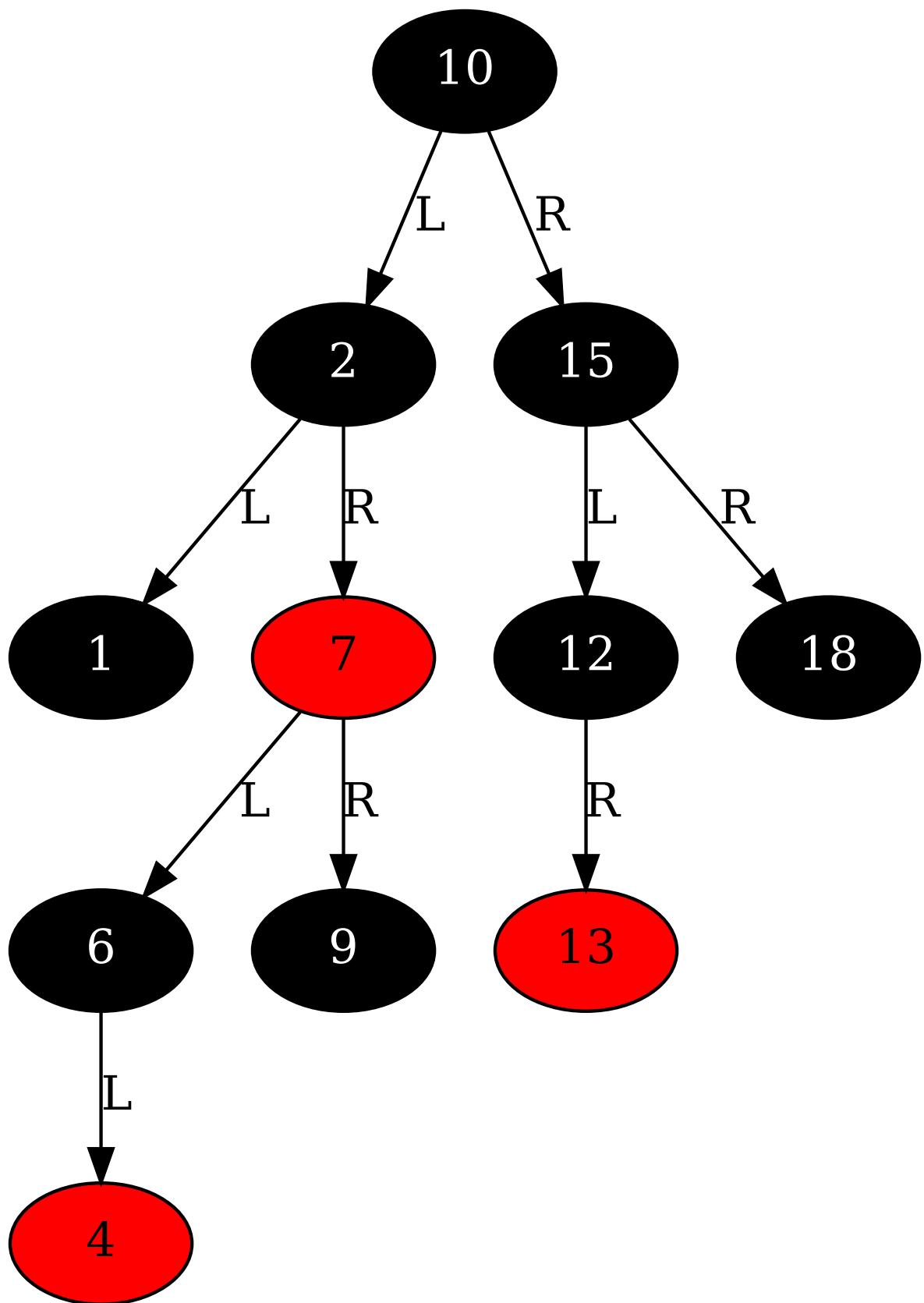
Inserting 6:



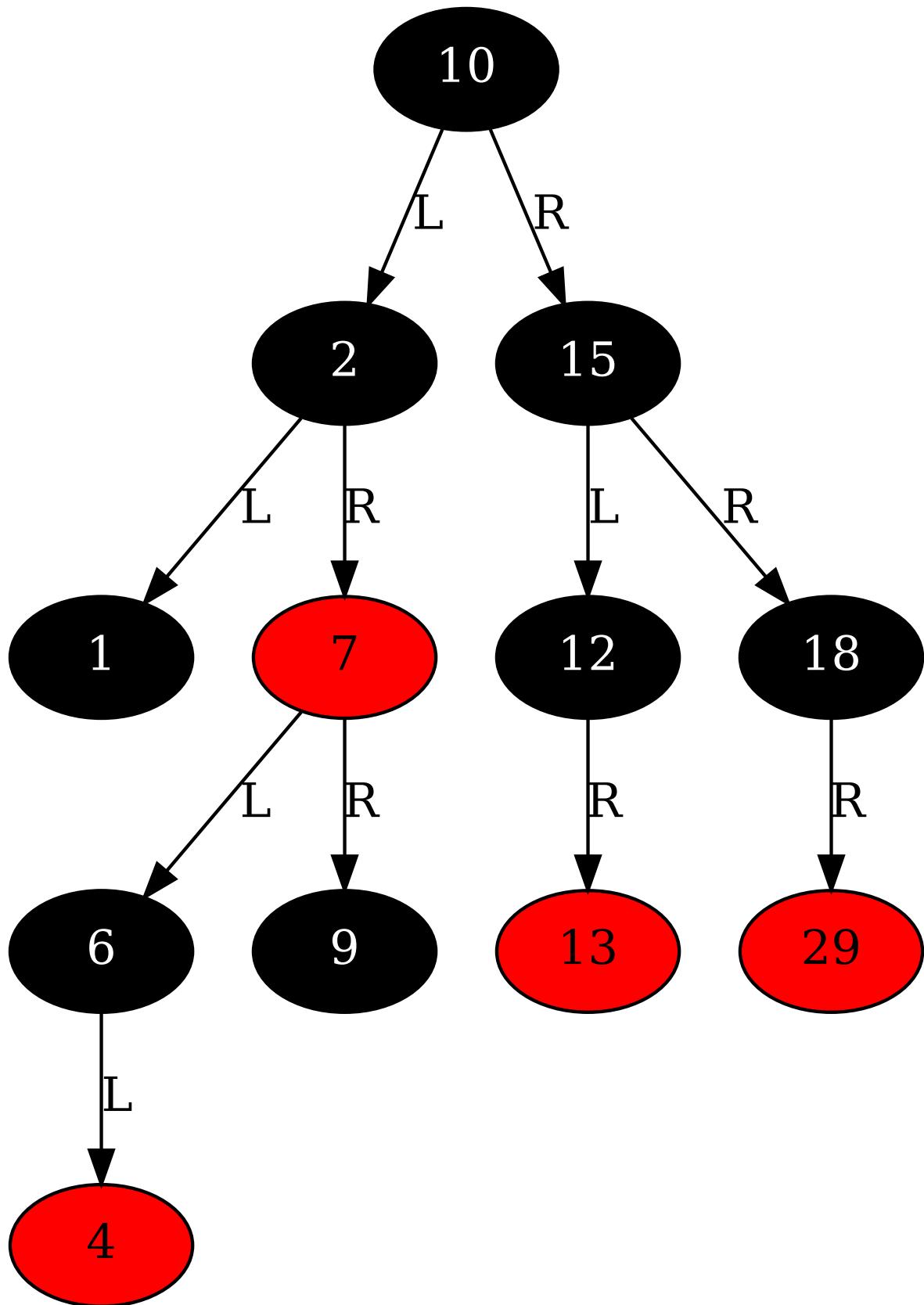
Inserting 7:

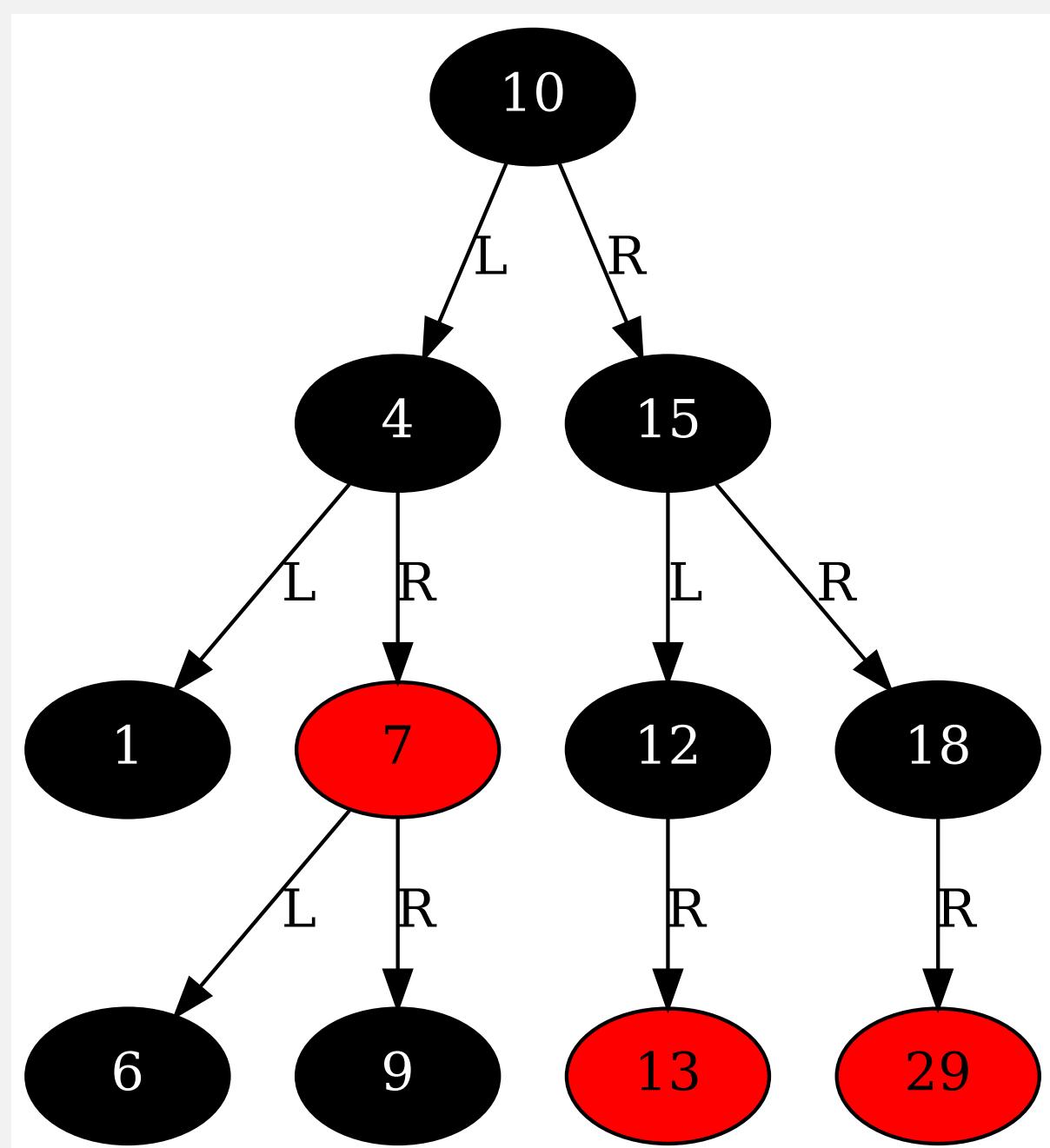


Inserting 4:

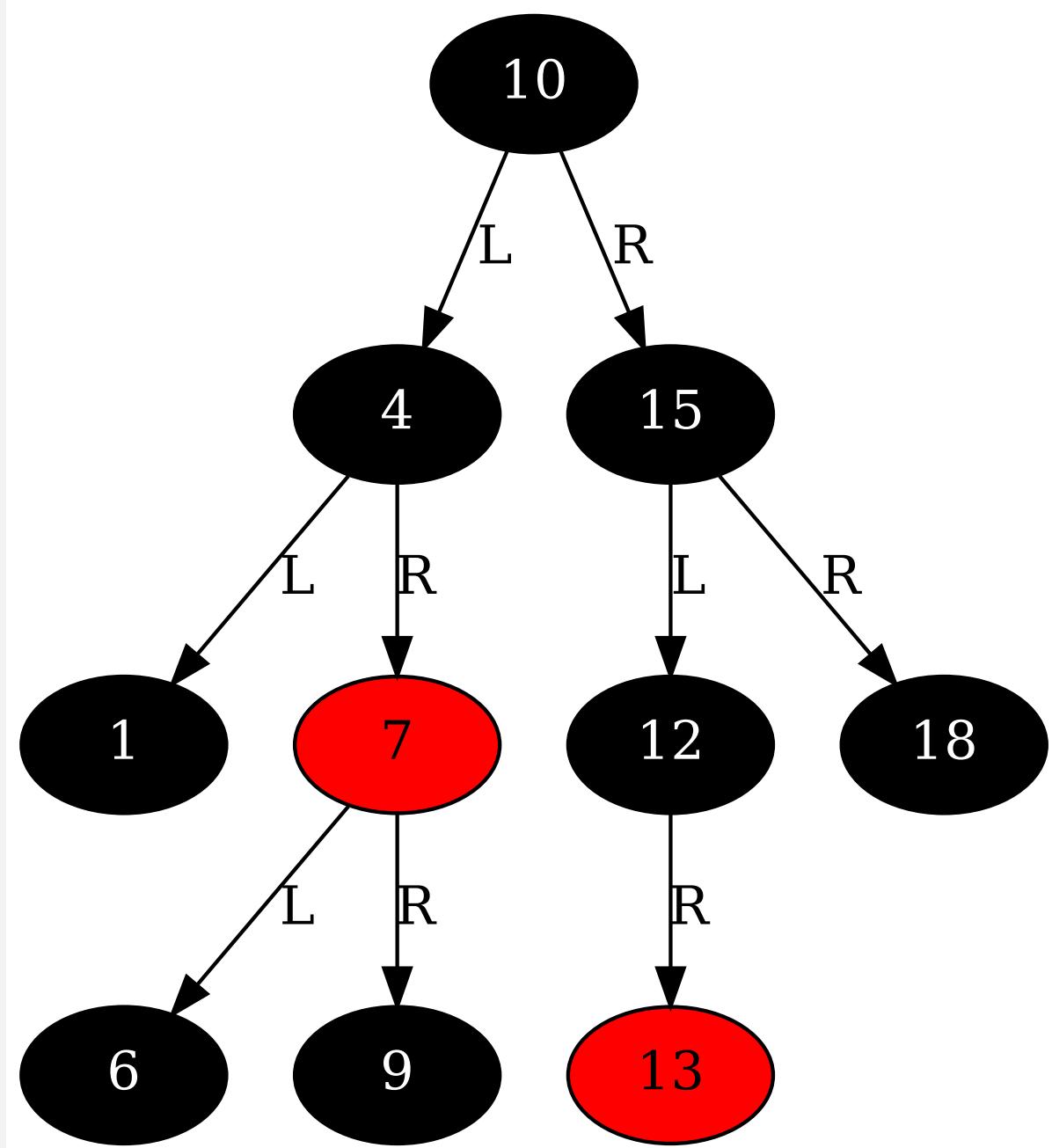


Inserting 29:

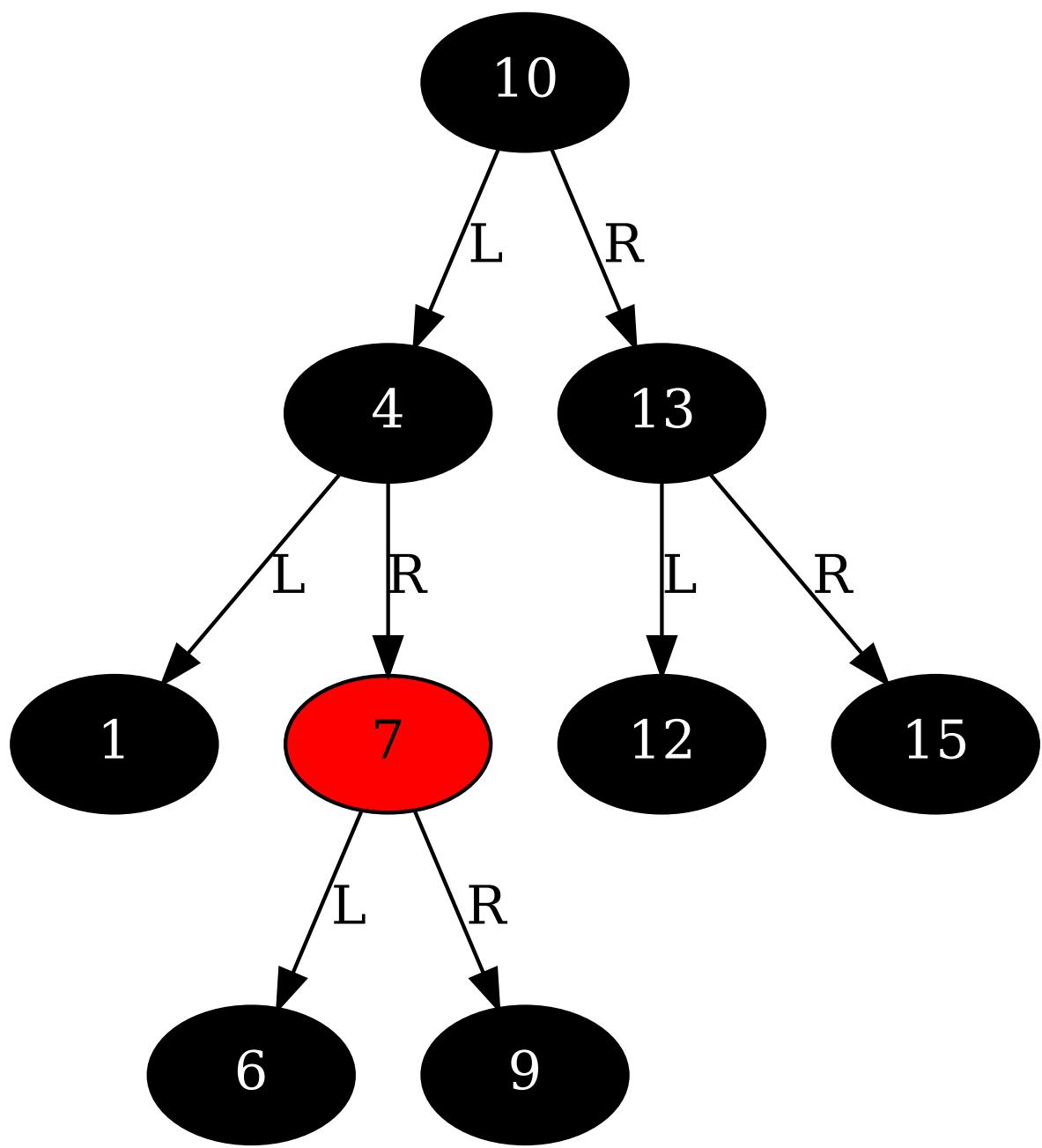




Deleting 29:



Deleting 18:



CELL 54

```
#Task 6
class BPlusTreeNode:
    def __init__(self, is_leaf=True):
        self.keys = []
        self.children = []
        self.is_leaf = is_leaf
        self.next = None # For leaf nodes
        self.parent = None

class BPlusTree:
    def __init__(self, order=3):
        self.order = order
        self.root = BPlusTreeNode(is_leaf=True)
        self.max_keys = order - 1

    def _find_leaf(self, key):
        """Find the leaf node where the key should be inserted"""
        current = self.root
        while not current.is_leaf:
            # Find the appropriate child to follow
            idx = 0
            while idx < len(current.keys) and key >= current.keys[idx]:
                idx += 1
            current = current.children[idx]
        return current

    def _find_leftmost_leaf(self):
        """Find the leftmost leaf in the tree"""
        current = self.root
        while not current.is_leaf:
            current = current.children[0]
        return current

    def insert(self, key):
        leaf = self._find_leaf(key)
        self._insert_into_leaf(leaf, key)

        if len(leaf.keys) > self.max_keys:
            self._split_leaf(leaf)

    def _insert_into_leaf(self, node, key):
        # Find position to insert
        pos = 0
        while pos < len(node.keys) and key > node.keys[pos]:
            pos += 1
        node.keys.insert(pos, key)

    def _split_leaf(self, node):
        mid = len(node.keys) // 2
        new_node = BPlusTreeNode(is_leaf=True)

        # Split keys
        new_node.keys = node.keys[mid:]
        node.keys = node.keys[:mid]

        # Update sibling pointers
        new_node.next = node.next
        node.next = new_node

        # Push up middle key to parent
        self._insert_into_parent(node, new_node.keys[0], new_node)

    def _insert_into_parent(self, left, key, right):
        if left == self.root:
```

```

# Create new root
new_root = BPlusTreeNode(is_leaf=False)
new_root.keys = [key]
new_root.children = [left, right]
self.root = new_root
left.parent = new_root
right.parent = new_root
return

parent = left.parent
# Find position to insert
pos = 0
while pos < len(parent.keys) and key > parent.keys[pos]:
    pos += 1

parent.keys.insert(pos, key)
parent.children.insert(pos+1, right)
right.parent = parent

if len(parent.keys) > self.max_keys:
    self._split_internal(parent)

def _split_internal(self, node):
    mid = len(node.keys) // 2
    mid_key = node.keys[mid]

    new_node = BPlusTreeNode(is_leaf=False)
    new_node.keys = node.keys[mid+1:]
    new_node.children = node.children[mid+1:]
    node.keys = node.keys[:mid]
    node.children = node.children[:mid+1]

    # Update parent pointers
    for child in new_node.children:
        child.parent = new_node

    # Push up middle key to parent
    self._insert_into_parent(node, mid_key, new_node)

def visualize(self):
    """Visualize B+ Tree"""
    dot = Digraph()
    self._add_nodes(self.root, dot)

    # Add horizontal links for leaf nodes
    current = self._find_leftmost_leaf()
    while current and current.next:
        dot.edge(str(id(current)), str(id(current.next)), style="dashed",
constraint="false")
        current = current.next

    display(dot)
    return dot

def _add_nodes(self, node, dot):
    node_id = str(id(node))
    label = " | ".join(str(k) for k in node.keys) + " | "
    shape = "rectangle" if node.is_leaf else "ellipse"
    dot.node(node_id, label=label, shape=shape)

    if not node.is_leaf:
        for child in node.children:
            child_id = str(id(child))
            self._add_nodes(child, dot)
            dot.edge(node_id, child_id)

```

```
print("\n" + "="*50)
print("Task 6: B+ Tree Insertion (m=3)")
print("="*50)
strings = ["era", "ban", "bat", "kin",
           "day", "log", "rye", "max",
           "won", "ace", "ado", "bug",
           "cop", "gas", "let", "fax"]
bptree = BPlusTree(order=3)

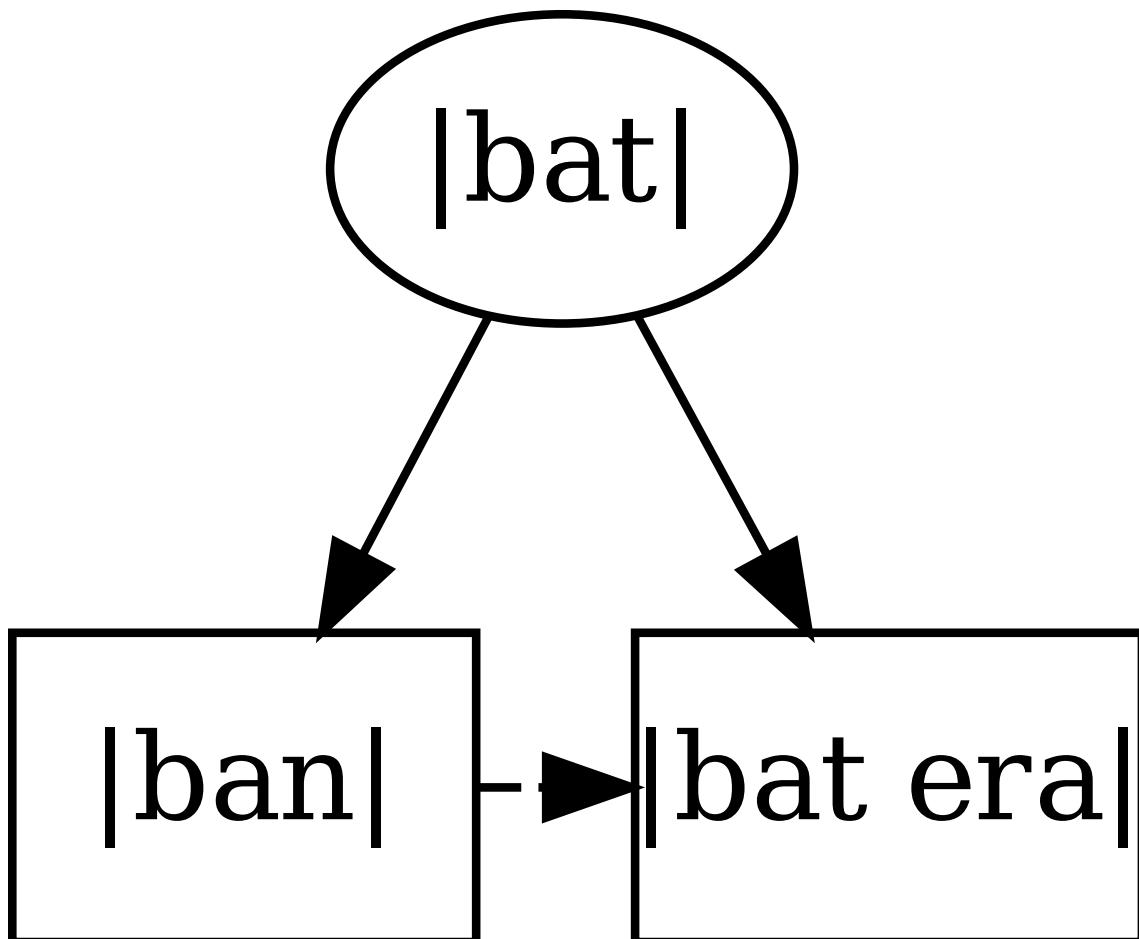
print("Insertion sequence:")
for i, s in enumerate(strings):
    bptree.insert(s)
    if (i+1) % 3 == 0:
        print(f"\nAfter {i+1} insertions:")
        bptree.visualize()

print("\nFinal B+ Tree:")
bptree.visualize()
```

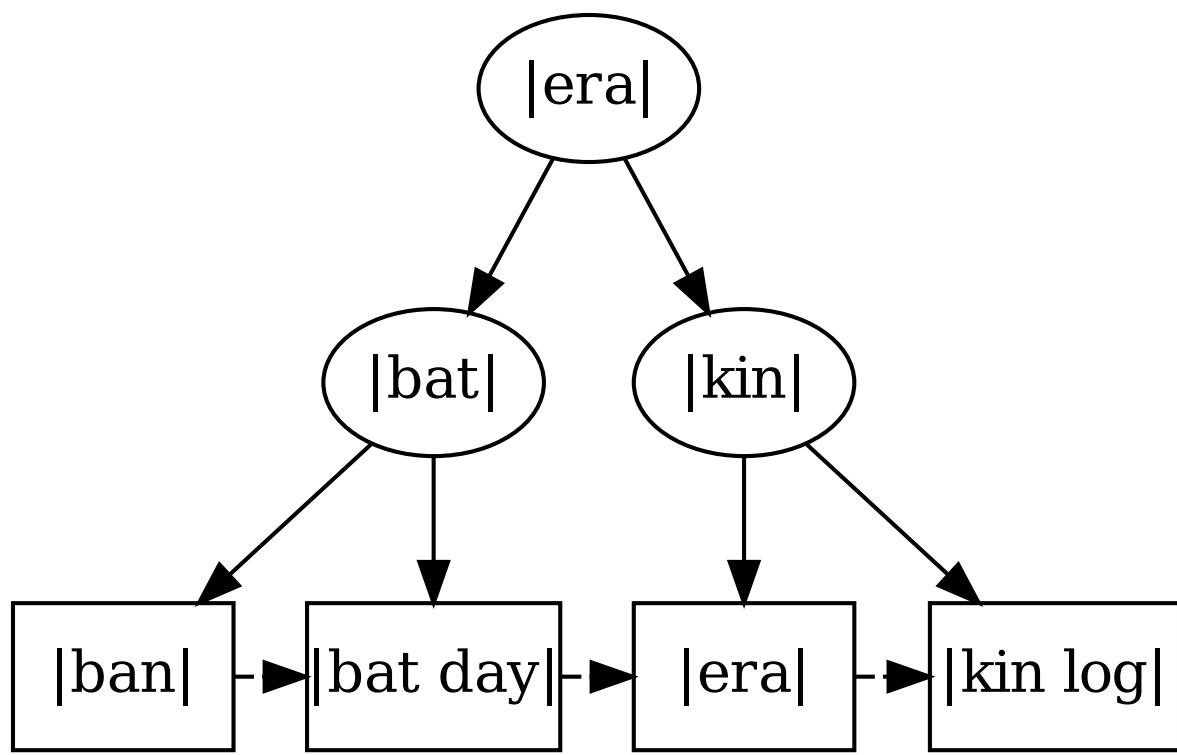
```
=====  
Task 6: B+ Tree Insertion (m=3)  
=====
```

Insertion sequence:

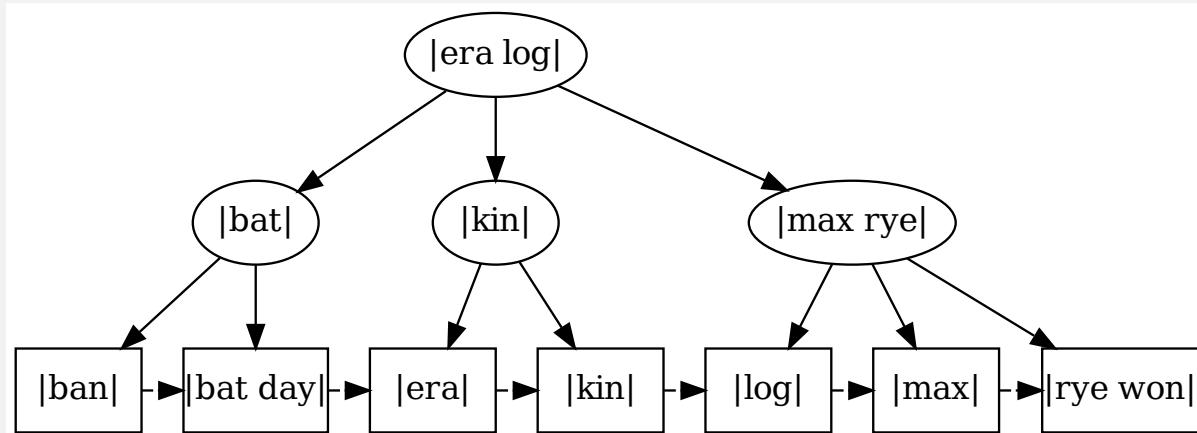
After 3 insertions:



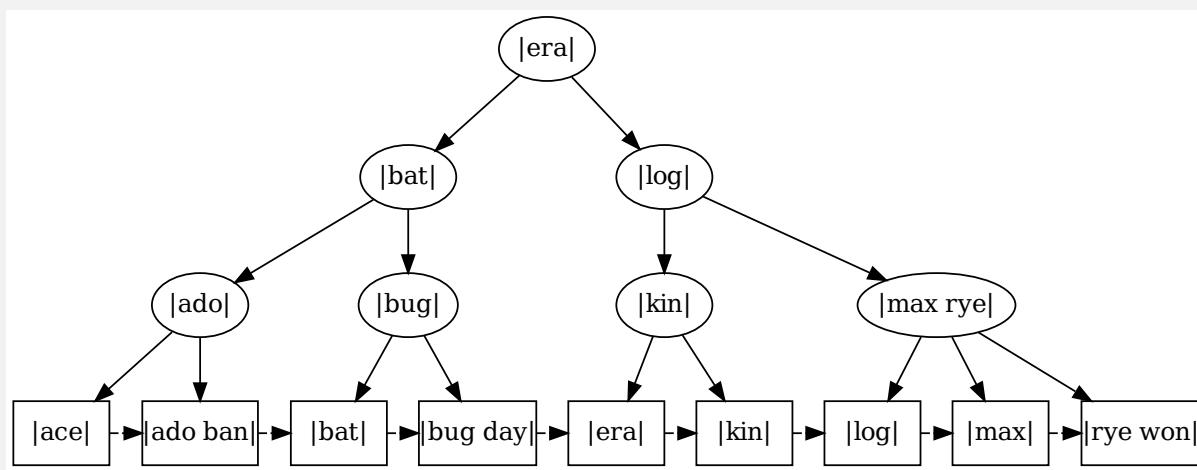
After 6 insertions:



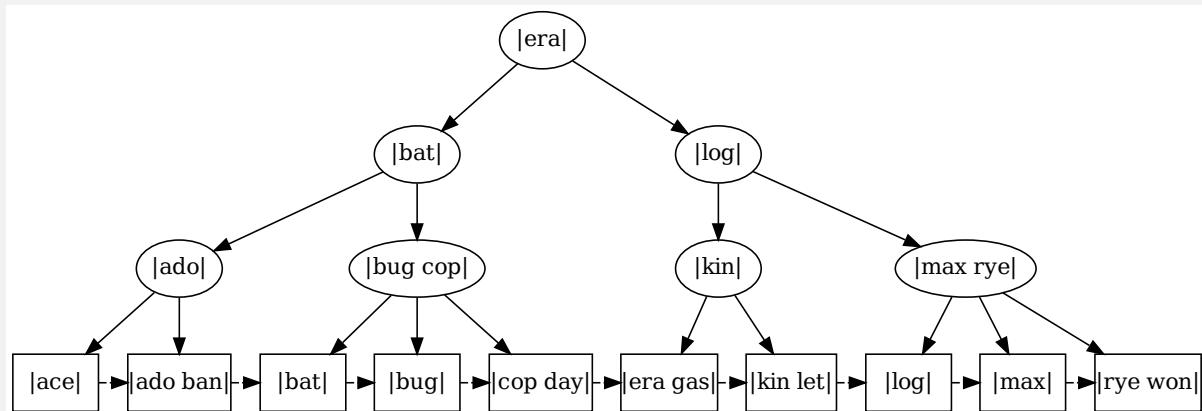
After 9 insertions:



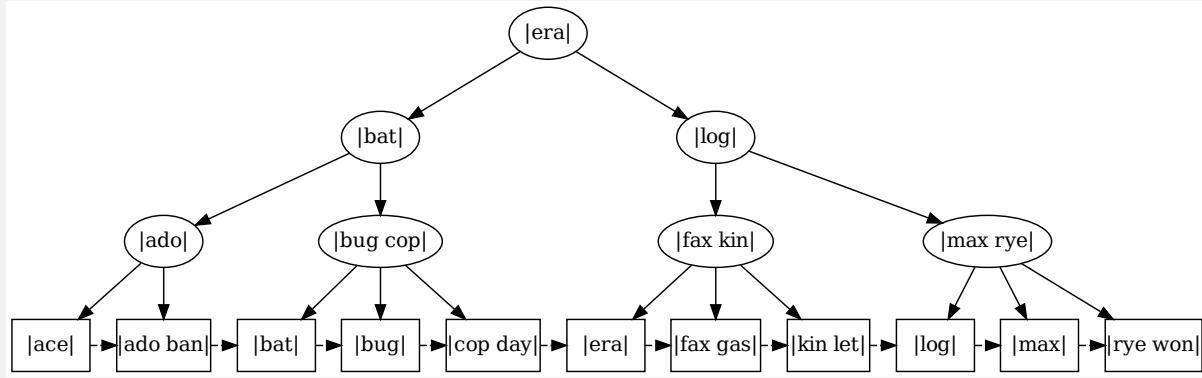
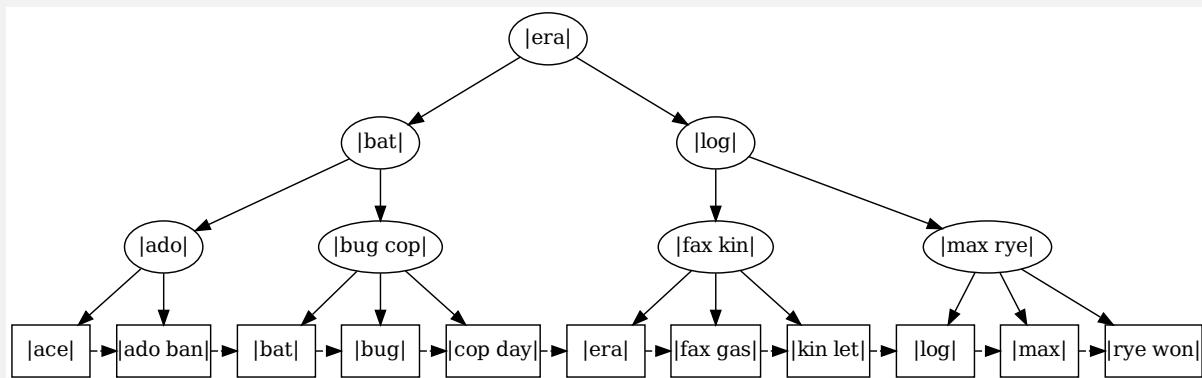
After 12 insertions:



After 15 insertions:



Final B+ Tree:



CELL 55

```

#Task 7
class BTreenode:
    def __init__(self):
        self.keys = []
        self.children = []
        self.parent = None # Adding a parent pointer

    def __str__(self):
        return " | ".join(str(k) for k in self.keys) + " | "

class BTREE:
    def __init__(self, order=3):
        self.order = order
        self.root = BTreenode()
        self.max_keys = order - 1
        self.min_keys = (order) // 2 # Minimum number of keywords

    def insert(self, key):
        # Find the insertion position starting from the root
        current = self.root
        path = [] # Recording paths for backtracking

        while current.children:
            i = len(current.keys) - 1
            while i >= 0 and key < current.keys[i]:
                i -= 1
            i += 1

            path.append((current, i))
            current = current.children[i]

        # Insert at the leaf node
        insort(current.keys, key)

        # Leaf node overflow
        "Process splits leaf → parent node → root node"
        while len(current.keys) > self.max_keys:
            if current == self.root:
                self._split_root()
                break
            else:
                current = self._split_node(current, path)
                if not path:
                    break
                current, idx = path.pop()

    def _split_root(self):
        new_root = BTreenode()
        left_child = self.root

        mid = (len(left_child.keys) - 1) // 2
        #mid = (len(left_child.keys)) // 2
        mid_key = left_child.keys[mid]

        # Create the right child node
        right_child = BTreenode()
        right_child.keys = left_child.keys[mid+1:]
        right_child.children = left_child.children[mid+1:] if left_child.children else []
        for child in right_child.children:
            child.parent = right_child

        # Left child
        left_child.keys = left_child.keys[:mid]

```

```
left_child.children = left_child.children[:mid+1] if left_child.children else []
# Set new root
new_root.keys = [mid_key]
new_root.children = [left_child, right_child]
left_child.parent = new_root
right_child.parent = new_root
self.root = new_root

def _split_node(self, node, path):
    # Get the parent node and the index of the current node
    parent, idx = path[-1] if path else (None, 0)
    if not parent:
        return node

    mid = (len(node.keys) - 1) // 2
    mid_key = node.keys[mid]

    # Create a new node
    new_node = BTTreeNode()
    new_node.keys = node.keys[mid+1:]
    new_node.children = node.children[mid+1:] if node.children else []
    for child in new_node.children:
        child.parent = new_node
    new_node.parent = parent
    node.keys = node.keys[:mid]
    node.children = node.children[:mid+1] if node.children else []
    parent.keys.insert(idx, mid_key)
    parent.children.insert(idx+1, new_node)

    return parent

def visualize(self):
    dot = Digraph()
    self._add_nodes(self.root, dot)
    display(dot)
    return dot

def _add_nodes(self, node, dot):
    if not node.keys: # Skip empty nodes
        return

    node_id = str(id(node))
    label = "| " + "|".join(f"[{k}]" for k in node.keys) + " |"
    dot.node(node_id, label=label)

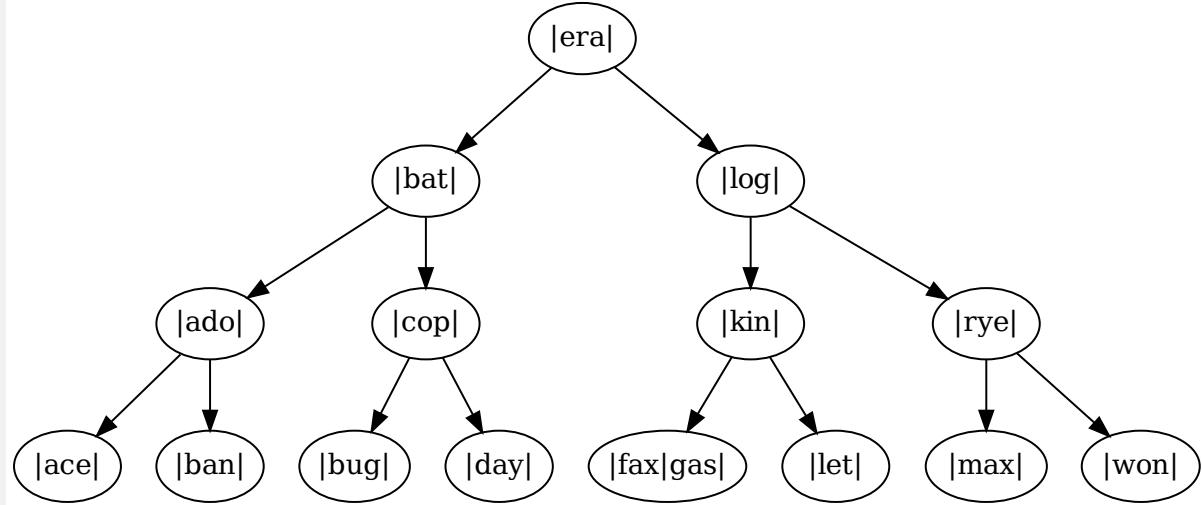
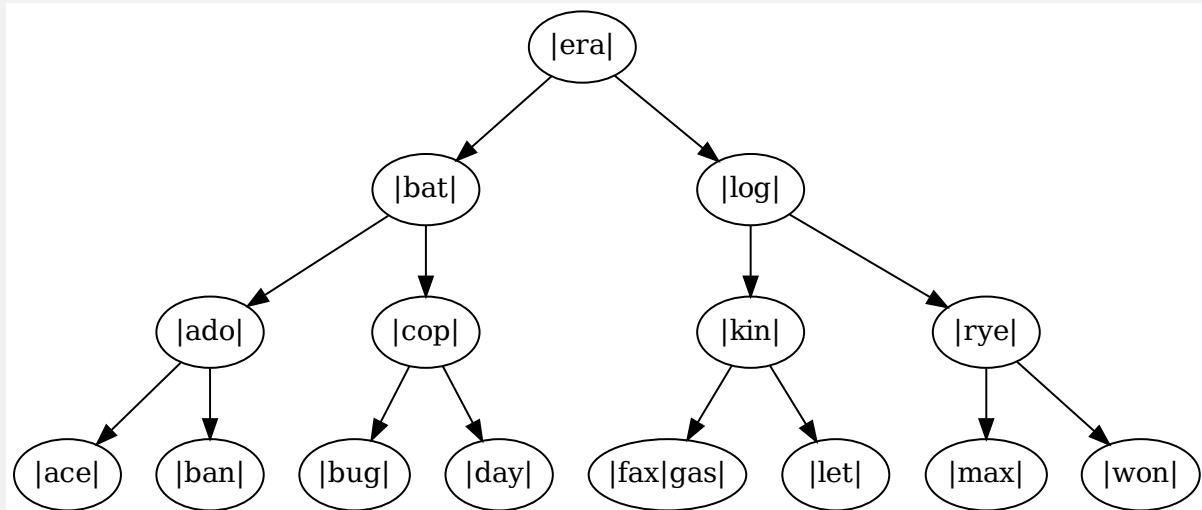
    for child in node.children:
        if child.keys: # Only add child nodes with keywords
            child_id = str(id(child))
            self._add_nodes(child, dot)
            dot.edge(node_id, child_id)

# using task 6 values
strings = ["era", "ban", "bat", "kin", "day", "log",
           "rye", "max", "won", "ace", "ado", "bug",
           "cop", "gas", "let", "fax"]

btree = BTTree(order=3)
for s in strings:
    btree.insert(s)

print("B Tree (m=3):")
btree.visualize()
```

B Tree (m=3):



CELL 56

```
#Task 9

function range_query_bplus_tree(root, start_key, end_key):

    // 1. Find the leaf node containing the start key
    current_node = root

    while current_node is not leaf node:
        // Find the first key greater than or equal to start_key

        in the current node
        index = 0

        while index < current_node.key_count and start_key > current_node.keys[index]:
            index = index + 1
            current_node = current_node.children[index]

    // 2. Initialize the result set

    results = empty list

    // 3. Traverse the leaf nodes until the end key

    while current_node is not null:
        for each key in current_node.keys:
            if key >= start_key and key <= end_key:
                add key to results
            else if key > end_key:
                return results // Out of range, end query

        // Move to the next leaf node
        current_node = current_node.next

    return results
```

```
[ESC] [1;36m File [ESC] [1;32m"<ipython-input-247-fbc4abc813a4>"[ESC] [1;36m, line
[ESC] [1;32m3[ESC] [0m
[ESC] [1;33m     function range_query_bplus_tree(root, start_key, end_key):[ESC] [0m
[ESC] [1;37m             ^[ESC] [0m
[ESC] [1;31mSyntaxError[ESC] [0m[ESC] [1;31m:[ESC] [0m invalid syntax
```

CELL 57

CELL 58

CELL 59

CELL 60

CELL 61

CELL 62

CELL 63

CELL 64

CELL 65

CELL 66

CELL 67

CELL 68

6 Advanced Hashing and Sorting

Advanced Hashing and Sorting

Date	Author	Comment
2025/08/01 20:26	Yi Guan	Ready to Mark
2025/08/01 20:26	Yi Guan	hihi
2025/08/03 22:59	Yi Guan	Ready to Mark
2025/08/03 22:59	Yi Guan	I fixed an issue in Task 2 where the bucket size exceeded the maximum limit, and also improved the code overall. In Task 4, I added more comments and 100 randomly generated data sets to demonstrate the merging process and final results.
2025/08/03 22:59	Yi Guan	I hope this meets the requirements of the task.
2025/08/03 22:59	Yi Guan	:smiling_face_with_tear:
2025/08/04 11:31	Xiangwen Yang	Good work, but it still has a minor issue with the merge sort. After the merge, it should be a "writing list" that contains all the sorted values.
2025/08/04 11:31	Xiangwen Yang	Discuss
2025/08/04 16:52	Yi Guan	Got it, sir. I have fixed the issue. Now it should be printed as a list.
2025/08/04 16:52	Yi Guan	something like this
2025/08/04 16:53	Yi Guan	image comment
2025/08/08 20:12	Xiangwen Yang	Good work!
2025/08/08 20:12	Xiangwen Yang	Complete

DEAKIN UNIVERSITY

ADVANCED ALGORITHMS

ONTRACK SUBMISSION

Advanced Hashing and Sorting

Submitted By:

Yi GUAN

ppv

2025/08/04 16:52

Tutor:

Xiangwen YANG

August 4, 2025



Summary of Advanced Hashing and Sorting

Key k is placed into a bucket using the simplest approach: the bucket number is the remainder of k divided by the total number of buckets, N . The number of buckets is fixed from the outset; if a bucket is full, the entire structure must be rebuilt. If two keys collide, they can be linked together using a linked list, or they can be searched one by one to find an empty space.

Another approach is to avoid hard coding the number of buckets: buckets can be added or subtracted at any time as the amount of data increases, eliminating the need for a complete reshuffle and restart. This is how scalable hashing works, used by file systems like ZFS, GFS, and Spad FS. It still finds the data in one step and can grow almost as large as desired.

If the data is too large to fit in memory, an external sort is used. The simplest way to regress and merge versions uses only three caches, resulting in a total read and write count of approximately $2 \times \text{number of pages} \times (1 + \log_2 \text{number of pages})$. Every page is read and written once during each pass through the file.

If the data already has a B^+ -tree index based on the sort key, there's no need for a dedicated sort. When the index is "in order," the data pages are already largely sorted. Simply scanning the leaf level and retrieving the pages in order can help save a lot of extra reads and writes. Conversely, if the index is "random," with data pages scattered all over the place, retrieving records in key order may require a random jump for each record, which is prohibitively expensive and generally not cost-effective.

Expected Deliverables:

- Python code with clear documentation.
- Evidence of how the merge logic handles limited memory (buffer) constraints.
- Optional: log or print statements to visualize each merge pass.

Buffer Usage Limits:

The buffer size is n pages, but only $n-1$ pages can be used to read input runs. The last buffer page must be reserved for writing output. This limitation forces us to merge at most $n-1$ runs at a time.

Code: `merge_degree = self.buffer_size - 1`

Why use $n-1$ pages for reading:

If the buffer size is 3, only 2 runs can be merged at a time. The third buffer is needed to temporarily store the merged output before it is flushed to disk. This avoids data loss and simulates real disk I/O.

If all n pages were used for reading, there would be no space to store the merged results, resulting in data loss or overwriting of previously written data.

Merge Phase Logic:

Each merge merges multiple groups of runs using a priority queue. This process repeats multiple times until all data is merged into a single sorted run. Each merge adheres to the buffer limit, grouping at most $n-1$ runs at a time.

Eg. Degree = `buffer_size -1 =2`

First round: 100 -> 50

Second round: 50 -> 25

Third round: 25 -> 13

Forth round: 13 -> 7

Fifth round: 7 -> 4

Sixth round: 4 -> 2

Seventh round: 2 -> 1

I/O Count: The system simulates I/O operations (`simulate_read` and `simulate_write`) to simulate real disk costs, which is crucial for external sorting.

Code:

```
def simulate_read(self):  
    self.io_count += 1 # Simulated disk read  
  
def simulate_write(self):  
    self.io_count += 1 # Simulated disk write
```

CELL 01

```
import numpy as np
import os
import string
import random
import csv
from collections import defaultdict
import heapq
import math
import hashlib
from itertools import chain
```

1 Core Functions

CELL 03

```
def hash_funtion(key):

    return '{0:016b}'.format(key)

class Bucket:

    def __init__(self,local_depth,index,empty_spaces,id):

        self.id = id
        self.local_depth = local_depth
        self.index = index
        self.empty_spaces = empty_spaces

class Directory:

    def __init__(self,global_depth,directory_records):

        self.global_depth = global_depth,
        self.directory_records = directory_records

class DirectoryRecord:

    def __init__(self,bucket, hash_prefix):

        self.hash_prefix = hash_prefix
        self.value = bucket
```

2 Hyper Parameters

CELL 05

```
bucket_capacity = 2
bucket_number = 3
global_depth = 1
```

CELL 06

```
# Initialization of buckets
bucket1 = Bucket(local_depth = 1, empty_spaces = bucket_capacity, index = [], id = 1)
bucket2 = Bucket(local_depth = 1, empty_spaces = bucket_capacity, index = [], id = 2)

# Initialization of directory
directory_records = list()
```

```
directory_records.append(DirectoryRecord(hash_prefix = 0, bucket = bucket1))
directory_records.append(DirectoryRecord(hash_prefix = 1, bucket = bucket2))

directory = Directory(global_depth = 1, directory_records = directory_records)
```

```
directory_records
```

CELL 07

```
[<__main__.DirectoryRecord at 0x25e3cdcd30>,
 <__main__.DirectoryRecord at 0x25e3cdcd8b0>]
```

3 Insertion Algorithm

CELL 09

```

def insert(index):

    global directory
    global bucket_number

    t_id = index[0]
    hash_key = hash_funtion(int(t_id))

    hash_prefix = int(hash_key[-directory.global_depth[0]:], 2)

    bucket = directory.directory_records[hash_prefix].value
    bucket.index.append(index)
    bucket.empty_spaces = int(bucket.empty_spaces)-1

    if(bucket.empty_spaces < 0):

        temporary_memory = bucket.index
        bucket.empty_spaces = bucket_capacity
        bucket.index = []

        if (directory.global_depth[0] > bucket.local_depth):

            # NUMBER OF LINKED BUCKETS
            number_of_links = 2***(directory.global_depth[0] - bucket.local_depth)
            bucket.local_depth = bucket.local_depth + 1
            number_of_modify_links = number_of_links/2

            new_bucket = Bucket(local_depth = bucket.local_depth, index=[], empty_spaces =
bucket_capacity, id = bucket_number)

            for directory_record in directory.directory_records:

                if(directory_record.value == bucket):
                    if(number_of_modify_links != 0):
                        number_of_modify_links = number_of_modify_links - 1
                    else:
                        directory_record.value = new_bucket
                        bucket_number = bucket_number + 1

                for i in range(len(temporary_memory)):
                    insert(temporary_memory[i])

            elif (directory.global_depth[0] == bucket.local_depth):

                new_directory_len = 2 * len(directory.directory_records)
                new_directory_records = []

                for directory_record_number in range(new_directory_len):
                    new_directory_records.append(DirectoryRecord(hash_prefix=directory_record_
number,bucket=Bucket(local_depth=1,index=[],empty_spaces=bucket_capacity,id=bucket_num
ber)))
                bucket_number = bucket_number + 1

                new_directory = Directory(global_depth=directory.global_depth[0]+1,directory_r
ecords=new_directory_records)

            # REHASING

            for directory_record in directory.directory_records:
                haskey1 = '0'+hash_funtion(directory_record.hash_prefix)
                haskey2 = '1'+hash_funtion(directory_record.hash_prefix)

```

```
new_index1 = int(haskey1[-directory.global_depth[0]:],2)
new_index2 = int(haskey2[-directory.global_depth[0]:],2)

new_directory.directory_records[new_index1].value = directory_record.value
new_directory.directory_records[new_index2].value = directory_record.value

directory= new_directory

for i in range(len(tempopary_memory)):

    insert(tempopary_memory[i],lock)
```

4 Main

```
t_id = 0
t_amount = 100
u_name = 'David'

insert([t_id, t_amount, u_name])
```

CELL 11

CELL 12

CELL 13

CELL 14

5 Task 2

```
class Bucket:
    _id_counter = 0

    def __init__(self, local_depth):
        self.id = Bucket._id_counter
        Bucket._id_counter += 1
        self.local_depth = local_depth
        self.data = []

    def is_full(self):
        return len(self.data) >= 3 # Bucket size limit is 3

    def __str__(self):
        return f"[B{self.id} depth={self.local_depth}] {self.data}"


class Directory:
    def __init__(self):
        self.global_depth = 1
        self.size = 2 ** self.global_depth
        self.buckets = {}
        self.directory = []

        # Initialize two buckets
        b0 = Bucket(local_depth=1)
        b1 = Bucket(local_depth=1)
        self.buckets[b0.id] = b0
        self.buckets[b1.id] = b1
        self.directory = [b0.id, b1.id]

    def _calculate_ascii_hash(self, value):
        "Calculate the sum of ASCII values of the string"
        str_val = str(value)
        ascii_sum = sum(ord(char) for char in str_val)
        return ascii_sum
```

CELL 16

```
def get_bucket_index(self, hash_value):
    """Get directory index"""
    return hash_value & ((1 << self.global_depth) - 1)

def get_bucket(self, h):
    bucket_id = self.directory[h]
    return self.buckets[bucket_id]

def double_directory(self):
    # Double the directory
    new_directory = self.directory * 2
    self.directory = new_directory
    self.global_depth += 1
    self.size = len(self.directory)
    print(f"  >> Directory doubled to size {self.size}, global depth now
{self.global_depth}")
    return True

def can_split_bucket(self, bucket):
    "Check if bucket can be split by deeper hash"
    if len(bucket.data) < 2:
        return False

    # Check if values have different prefixes at next depth
    new_depth = bucket.local_depth + 1
    mask = (1 << new_depth) - 1
    prefixes = set()

    for val in bucket.data:
        h = self._calculate_ascii_hash(val)
        prefix = h & mask
        prefixes.add(prefix)

    # If different prefixes found, split is possible
    if len(prefixes) > 1:
        return True

    return False # All values share the same prefix

def split_bucket(self, bucket_id):
    old_bucket = self.buckets[bucket_id]

    # Check if split is possible
    if not self.can_split_bucket(old_bucket):
        print(f"  !! Cannot split B{old_bucket.id} - all keys have same prefix at next
depth")
        return False

    old_depth = old_bucket.local_depth
    old_bucket.local_depth += 1

    # Double directory if needed
    if old_bucket.local_depth > self.global_depth:
        self.double_directory()

    # Create new bucket
    new_bucket = Bucket(local_depth=old_bucket.local_depth)
    self.buckets[new_bucket.id] = new_bucket

    # Update directory pointers
    high_bit = 1 << (old_bucket.local_depth - 1)
    for i in range(len(self.directory)):
        if self.directory[i] == bucket_id:
            # Use high bit to decide bucket
            if i & high_bit:
                self.directory[i] = new_bucket.id
```

```
# Re-distribute old data
temp = old_bucket.data[:]
old_bucket.data.clear()

for val in temp:
    h = self._calculate_ascii_hash(val)
    # Use new depth to get bucket
    bucket_index = h & ((1 << self.global_depth) - 1)
    b = self.get_bucket(bucket_index)
    b.data.append(val)

print(f"  >> Split B{bucket_id} into B{bucket_id} and B{new_bucket.id}")
return True

def insert(self, value):
    # Compute ASCII hash
    h = self.get_bucket_index(self._calculate_ascii_hash(value))
    bucket = self.get_bucket(h)
    str_val = str(value)
    ascii_sum = self._calculate_ascii_hash(value)
    print(f"  Inserting {value} ('{str_val}'->ASCII:{ascii_sum}) → index:{h}")

    if value in bucket.data:
        print(f"  !! Duplicate value {value} ignored.")
        return

    # If bucket full, try to split
    if bucket.is_full():
        print(f"  !! Bucket B{bucket.id} full. Attempting to split...")
        if not self.split_bucket(bucket.id):
            print(f"  !! ERROR: Cannot insert {value}. Bucket B{bucket.id} is full and cannot be split.")
            return

    else:
        # Get bucket again after split
        h = self.get_bucket_index(self._calculate_ascii_hash(value))
        bucket = self.get_bucket(h)

    # Insert value
    bucket.data.append(value)
    print(f"  >> Inserted {value} into B{bucket.id}")

def __str__(self):
    result = f"\n--- Directory (global depth = {self.global_depth}, size = {self.size}) ---\n"
    seen_buckets = set()

    # Show directory entries
    for i, b_id in enumerate(self.directory):
        # Show binary index
        binary_index = bin(i)[2:].zfill(self.global_depth)[-self.global_depth:]

        if b_id not in seen_buckets:
            result += f"  {binary_index} → {self.buckets[b_id]}\n"
            seen_buckets.add(b_id)
        else:
            # Reference to existing bucket
            result += f"  {binary_index} → (ref:B{b_id})\n"

    # Show all buckets
    result += "\nAll Buckets:\n"
    for bucket_id, bucket in self.buckets.items():
        if bucket_id not in seen_buckets:
            result += f"  {bucket}\n"
```

```
    return result

def main():
    values = [16, 22, 26, 20, 3, 1, 12, 91, 28, 26, 47, 11, 13, 19, 38, 47, 46]
    directory = Directory()

    for i, val in enumerate(values):
        print(f"\nStep {i + 1}: Insert {val}")
        directory.insert(val)
        print(directory)

if __name__ == "__main__":
    main()
```

```
Step 1: Insert 16
Inserting 16 ('16'→ASCII:103) → index:1
>> Inserted 16 into B1
```

```
--- Directory (global depth = 1, size = 2) ---
0 → [B0 depth=1] []
1 → [B1 depth=1] [16]
```

All Buckets:

```
Step 2: Insert 22
Inserting 22 ('22'→ASCII:100) → index:0
>> Inserted 22 into B0
```

```
--- Directory (global depth = 1, size = 2) ---
0 → [B0 depth=1] [22]
1 → [B1 depth=1] [16]
```

All Buckets:

```
Step 3: Insert 26
Inserting 26 ('26'→ASCII:104) → index:0
>> Inserted 26 into B0
```

```
--- Directory (global depth = 1, size = 2) ---
0 → [B0 depth=1] [22, 26]
1 → [B1 depth=1] [16]
```

All Buckets:

```
Step 4: Insert 20
Inserting 20 ('20'→ASCII:98) → index:0
>> Inserted 20 into B0
```

```
--- Directory (global depth = 1, size = 2) ---
0 → [B0 depth=1] [22, 26, 20]
1 → [B1 depth=1] [16]
```

All Buckets:

```
Step 5: Insert 3
Inserting 3 ('3'→ASCII:51) → index:1
>> Inserted 3 into B1
```

```
--- Directory (global depth = 1, size = 2) ---
0 → [B0 depth=1] [22, 26, 20]
1 → [B1 depth=1] [16, 3]
```

All Buckets:

```
Step 6: Insert 1
Inserting 1 ('1'→ASCII:49) → index:1
>> Inserted 1 into B1
```

```
--- Directory (global depth = 1, size = 2) ---
0 → [B0 depth=1] [22, 26, 20]
1 → [B1 depth=1] [16, 3, 1]
```

All Buckets:

```
Step 7: Insert 12
Inserting 12 ('12'→ASCII:99) → index:1
!! Bucket B1 full. Attempting to split...
>> Directory doubled to size 4, global depth now 2
>> Split B1 into B1 and B2
>> Inserted 12 into B2

--- Directory (global depth = 2, size = 4) ---
00 → [B0 depth=1] [22, 26, 20]
01 → [B1 depth=2] [1]
10 → (ref:B0)
11 → [B2 depth=2] [16, 3, 12]
```

All Buckets:

```
Step 8: Insert 91
Inserting 91 ('91'→ASCII:106) → index:2
!! Bucket B0 full. Attempting to split...
>> Split B0 into B0 and B3
>> Inserted 91 into B3

--- Directory (global depth = 2, size = 4) ---
00 → [B0 depth=2] [22, 26]
01 → [B1 depth=2] [1]
10 → [B3 depth=2] [20, 91]
11 → [B2 depth=2] [16, 3, 12]
```

All Buckets:

```
Step 9: Insert 28
...
0000 → [B0 depth=2] [22, 26, 13]
0001 → [B1 depth=2] [1]
0010 → [B3 depth=2] [20, 91, 28]
0011 → [B2 depth=4] [3, 12]
0100 → (ref:B0)
0101 → (ref:B1)
0110 → (ref:B3)
0111 → [B4 depth=3] [16]
1000 → (ref:B0)
1001 → (ref:B1)
1010 → (ref:B3)
1011 → [B5 depth=4] [47, 38]
1100 → (ref:B0)
1101 → (ref:B1)
1110 → (ref:B3)
1111 → (ref:B4)
```

All Buckets:

6 Task 3

CELL 18

```

# Core Classes and Functions
class Bucket:
    def __init__(self, local_depth, bucket_id):
        self.local_depth = local_depth # Number of bits used for bucket addressing
        self.data = [] # Values stored in this bucket
        self.id = bucket_id # Unique identifier for the bucket

    def is_full(self, capacity):
        return len(self.data) >= capacity # Check if bucket has reached its capacity

    def __str__(self):
        return f"B{self.id}(l={self.local_depth}): {sorted(self.data)}"

class Directory:
    #Manages the directory and buckets of the extendible hash table

    def __init__(self, global_depth):
        self.global_depth = global_depth # Number of bits used for directory addressing
        self.size = 2 ** global_depth # Current directory size
        # Initialize buckets with unique IDs
        self.buckets = [Bucket(global_depth, i) for i in range(self.size)]
        self.next_bucket_id = self.size # Track next available bucket ID

    def get_bucket(self, hash_value):
        index = hash_value % self.size # Determine bucket index
        return self.buckets[index]

    def double_directory(self):
        print(" Doubling directory size")
        self.global_depth += 1
        new_size = 2 ** self.global_depth

        # Create new directory with doubled size
        new_buckets = []
        for i in range(new_size):
            # Point to existing buckets (copy pointers)
            old_index = i % (new_size // 2)
            new_buckets.append(self.buckets[old_index])

        self.size = new_size
        self.buckets = new_buckets

    def split_bucket(self, bucket_idx):
        old_bucket = self.buckets[bucket_idx]
        print(f" Splitting bucket B{old_bucket.id}")

        # Create new bucket with incremented local depth
        new_bucket = Bucket(old_bucket.local_depth + 1, self.next_bucket_id)
        self.next_bucket_id += 1

        # Update local depths
        old_bucket.local_depth += 1
        new_bucket.local_depth = old_bucket.local_depth

        # Update directory pointers using high bit
        high_bit_mask = 1 << (old_bucket.local_depth - 1)
        for i in range(self.size):
            if self.buckets[i] == old_bucket and (i & high_bit_mask):
                self.buckets[i] = new_bucket

        return new_bucket

```

```
def __str__(self):
    result = f"Global Depth: {self.global_depth}\nDirectory Size: {self.size}\n"
    for i, bucket in enumerate(self.buckets):
        result += f"{i} -> {bucket}\n"
    return result

def hash_function(value):
    # Convert to string and sum ASCII values of characters
    return sum(ord(c) for c in str(value))

# Hyper Parameters
BUCKET_CAPACITY = 3
INITIAL_GLOBAL_DEPTH = 1

# Insertion Algorithm
def insert(directory, value):
    # Compute hash value
    hash_value = hash_function(value)
    bucket = directory.get_bucket(hash_value)

    print(f"\nInserting {value} (ASCII sum={hash_value})")

    # Check for duplicates
    if value in bucket.data:
        print(f"Duplicate value {value} ignored")
        print(directory)
        return

    # If bucket has space, insert directly
    if len(bucket.data) < BUCKET_CAPACITY:
        bucket.data.append(value)
        print(f"Added to B{bucket.id}")
        print(directory)
        return

    print(f"Bucket B{bucket.id} overflow! Splitting required")

    # Check if directory needs expansion
    if bucket.local_depth == directory.global_depth:
        directory.double_directory()
        # Recompute bucket after directory expansion
        hash_value = hash_function(value)
        bucket = directory.get_bucket(hash_value)

    # Split the overflowing bucket
    new_bucket = directory.split_bucket(bucket.id)
    print(f"Created new bucket B{new_bucket.id}")

    # Redistribute data from old bucket
    all_data = bucket.data + [value]
    bucket.data = []
    new_bucket.data = []

    # Reinsert all values
    for val in all_data:
        val_hash = hash_function(val)
        target_bucket = directory.get_bucket(val_hash)
        target_bucket.data.append(val)

    print(directory)

# Deletion Functionality
def delete(directory, value):
    hash_value = hash_function(value)
    bucket = directory.get_bucket(hash_value)
```

```
print(f"\nDeleting {value} (ASCII sum={hash_value})")

if value in bucket.data:
    bucket.data.remove(value)
    print(f"Removed from B{bucket.id}")
else:
    print(f"Value {value} not found")

print(directory)

# Statistics Function
def get_bucket_statistics(directory):
    """Calculate statistics about bucket utilization"""
    stats = {
        'total_buckets': len(set(directory.buckets)),
        'empty_buckets': 0,
        'average_utilization': 0.0,
        'min_utilization': BUCKET_CAPACITY,
        'max_utilization': 0
    }

    total_utilization = 0
    unique_buckets = set()

    for bucket in directory.buckets:
        # Only count each bucket once
        if id(bucket) not in unique_buckets:
            unique_buckets.add(id(bucket))
            utilization = len(bucket.data) / BUCKET_CAPACITY
            total_utilization += utilization

            if len(bucket.data) == 0:
                stats['empty_buckets'] += 1

            stats['min_utilization'] = min(stats['min_utilization'], utilization)
            stats['max_utilization'] = max(stats['max_utilization'], utilization)

    stats['average_utilization'] = total_utilization / stats['total_buckets'] if
    stats['total_buckets'] > 0 else 0
    return stats

# File Loading Function (New Feature)
def load_from_file(directory, filename):
    print(f"\nLoading data from {filename}")
    try:
        with open(filename, 'r') as file:
            for line in file:
                values = line.strip().split(',')
                for value in values:
                    if value: # Skip empty strings
                        insert(directory, int(value))
    except FileNotFoundError:
        print(f"Error: File {filename} not found")
```

CELL 19

```
def main():
    print("===== EXTENDIBLE HASHING =====")

    # Case 1: Basic insertion sequence
    print("\n CASE 1: Basic Insertion")
    dir1 = Directory(INITIAL_GLOBAL_DEPTH)
    values = [16, 22, 26, 20, 3, 1, 12, 91, 28, 26, 47, 11, 13, 19, 38, 47, 46]
    for value in values:
        insert(dir1, value)
        print("=" * 50)

    # Case 2: Random data generation
    print("\n CASE 2: Random Data Generation")
    import random
    dir2 = Directory(INITIAL_GLOBAL_DEPTH)
    for _ in range(20):
        value = random.randint(1, 100)
        insert(dir2, value)
        print("=" * 50)

    # Case 3: Edge cases
    print("\n CASE 3: Edge Cases")
    dir3 = Directory(INITIAL_GLOBAL_DEPTH)

    # Insert duplicate
    insert(dir3, 10)
    insert(dir3, 10) # Duplicate

    # Insert empty (should do nothing)
    # insert(dir3, None) # Uncomment to test

    # Insert very large number
    insert(dir3, 999999)

    # Insert negative number
    insert(dir3, -5)

    # Case 4: Load from file
    print("\n CASE 4: Load from File")

    # Create test data file
    with open('test_data.txt', 'w') as f:
        f.write("5,10,15,20,25,30,35,40,45,50")

    dir4 = Directory(INITIAL_GLOBAL_DEPTH)
    load_from_file(dir4, 'test_data.txt')

    # Case 5: Deletion and statistics
    print("\n CASE 5: Deletion and Statistics")
    dir5 = Directory(INITIAL_GLOBAL_DEPTH)
    for value in [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]:
        insert(dir5, value)

    stats_before = get_bucket_statistics(dir5)
    print("\nStatistics before deletion:")
    print(stats_before)

    delete(dir5, 20)
    delete(dir5, 40)
    delete(dir5, 99) # Not found

    stats_after = get_bucket_statistics(dir5)
    print("\nStatistics after deletion:")
    print(stats_after)
```

```
if __name__ == "__main__":
    main()
```

```
===== EXTENDIBLE HASHING =====
```

```
CASE 1: Basic Insertion
```

```
Inserting 16 (ASCII sum=103)
```

```
Added to B1
```

```
Global Depth: 1
```

```
Directory Size: 2
```

```
0 -> B0(ld=1): []
```

```
1 -> B1(ld=1): [16]
```

```
=====
```

```
Inserting 22 (ASCII sum=100)
```

```
Added to B0
```

```
Global Depth: 1
```

```
Directory Size: 2
```

```
0 -> B0(ld=1): [22]
```

```
1 -> B1(ld=1): [16]
```

```
=====
```

```
Inserting 26 (ASCII sum=104)
```

```
Added to B0
```

```
Global Depth: 1
```

```
Directory Size: 2
```

```
0 -> B0(ld=1): [22, 26]
```

```
1 -> B1(ld=1): [16]
```

```
=====
```

```
Inserting 20 (ASCII sum=98)
```

```
Added to B0
```

```
Global Depth: 1
```

```
Directory Size: 2
```

```
0 -> B0(ld=1): [20, 22, 26]
```

```
1 -> B1(ld=1): [16]
```

```
=====
```

```
Inserting 3 (ASCII sum=51)
```

```
Added to B1
```

```
Global Depth: 1
```

```
Directory Size: 2
```

```
0 -> B0(ld=1): [20, 22, 26]
```

```
1 -> B1(ld=1): [3, 16]
```

```
=====
```

```
Inserting 1 (ASCII sum=49)
```

```
Added to B1
```

```
Global Depth: 1
```

```
Directory Size: 2
```

```
0 -> B0(ld=1): [20, 22, 26]
```

```
1 -> B1(ld=1): [1, 3, 16]
```

```
=====
```

```
Inserting 12 (ASCII sum=99)
```

```
Bucket B1 overflow! Splitting required
```

```
Doubling directory size
```

```
Splitting bucket B1
```

```
Created new bucket B2
```

```
Global Depth: 2
```

```
Directory Size: 4
```

```
0 -> B0(ld=1): [20, 22, 26]
1 -> B1(ld=2): [1]
2 -> B0(ld=1): [20, 22, 26]
3 -> B2(ld=2): [3, 12, 16]
```

```
=====
Inserting 91 (ASCII sum=106)
Bucket B0 overflow! Splitting required
Splitting bucket B0
Created new bucket B3
Global Depth: 2
Directory Size: 4
0 -> B0(ld=2): [22, 26]
1 -> B1(ld=2): [1]
2 -> B3(ld=2): [20, 91]
3 -> B2(ld=2): [3, 12, 16]
```

```
=====
Inserting 28 (ASCII sum=106)
Added to B3
Global Depth: 2
Directory Size: 4
0 -> B0(ld=2): [22, 26]
1 -> B1(ld=2): [1]
2 -> B3(ld=2): [20, 28, 91]
3 -> B2(ld=2): [3, 12, 16]
```

```
=====
Inserting 26 (ASCII sum=104)
Duplicate value 26 ignored
Global Depth: 2
...
```

```
Deleting 99 (ASCII sum=114)
Value 99 not found
Global Depth: 3
Directory Size: 8
0 -> B0(ld=2): [80]
1 -> B1(ld=3): [10, 90, 100]
2 -> B3(ld=2): [60]
3 -> B2(ld=2): [30, 70]
4 -> B0(ld=2): [80]
5 -> B4(ld=3): [50]
6 -> B3(ld=2): [60]
7 -> B2(ld=2): [30, 70]
```

```
Statistics after deletion:
{'total_buckets': 5, 'empty_buckets': 0, 'average_utilization': 0.5333333333333333,
 'min_utilization': 0.3333333333333333, 'max_utilization': 1.0}
```

CELL 20

7 Task 4

CELL 22

```

class ExternalMergeSort:
    def __init__(self, total_pages=100, page_size=4, buffer_size=3):
        self.total_pages = total_pages
        self.page_size = page_size
        self.buffer_size = buffer_size
        self.io_count = 0
        self.pass_count = 0
        self.data = []
        self.sorted_pages = [] # Store final sorted pages

    def generate_data(self):
        print(f"Generating {self.total_pages} pages:")
        self.data = [
            sorted(random.sample(range(1, 1000), self.page_size))
            for _ in range(self.total_pages)
        ]
        # Display all pages
        for i, page in enumerate(self.data):
            print(f"Page {i:02d}: {page}")

    def simulate_read(self):
        self.io_count += 1

    def simulate_write(self):
        self.io_count += 1

    def sort_phase(self):
        print("\n== SORT PHASE ==")
        print("Sorting each page individually")
        # Pages are already sorted when generated, but in practice sorting might be needed
        print("Sorting complete")

    def merge_runs(self, runs):
        heap = []
        iterators = []

        # Initialize heap
        for i, run in enumerate(runs):
            # Flatten run if it's nested
            flat_run = list(chain.from_iterable(run)) if any(isinstance(item, list) for item in run) else run
            iterator = iter(flat_run)
            try:
                first_item = next(iterator)
                heapq.heappush(heap, (first_item, i, iterator))
            except StopIteration:
                pass
        iterators.append(iterator)

        # Merge runs
        result = []
        current_page = []
        while heap:
            val, run_idx, iterator = heapq.heappop(heap)
            current_page.append(val)

```

```
# Check if page is full
if len(current_page) >= self.page_size:
    result.append(current_page)
    self.simulate_write() # Simulate disk write
    current_page = []

# Get next item
try:
    next_val = next(iterator)
    heapq.heappush(heap, (next_val, run_idx, iterator))
except StopIteration:
    pass

# Add last page if there are remaining items
if current_page:
    result.append(current_page)
    self.simulate_write()

return result

def merge_phase(self):
    print("\n== MERGE PHASE ==")
    # Initial runs: each page is a separate run
    runs = [[item for item in page] for page in self.data]
    merge_degree = self.buffer_size - 1 # Merge degree = buffer size - 1 (output
page)

    pass_num = 0
    while len(runs) > 1:
        pass_num += 1
        new_runs = []
        print(f"\nMERGE PASS {pass_num}: Merging {len(runs)} runs with buffer size
{self.buffer_size}")
        print(f" Merge degree: {merge_degree}, Runs to process: {len(runs)}")
        print(f" Total runs: {len(runs)}, Runs per batch: {merge_degree}, Batches:
{math.ceil(len(runs)/merge_degree)}")

        # Process runs in batches
        for i in range(0, len(runs), merge_degree):
            batch = runs[i:i+merge_degree]
            batch_num = i//merge_degree + 1
            print(f" Batch {batch_num}: Merging {len(batch)} runs")

        # Simulate read for each run in batch
        for run in batch:
            self.simulate_read()

        # Merge batch
        merged_run = self.merge_runs(batch)
        new_runs.append(merged_run)

    runs = new_runs
    print(f" Created {len(runs)} new runs")

    # Final result
    if runs:
        # Flatten final result
        self.sorted_pages = []
        for run in runs:
            if any(isinstance(item, list) for item in run):
                # Flatten if run contains nested lists
                flat_run = list(chain.from_iterable(run))
            else:
                flat_run = run

            # Paginate the flattened run
```

```
        for i in range(0, len(flat_run), self.page_size):
            page = flat_run[i:i+self.page_size]
            self.sorted_pages.append(page)
        print(f"\nMerge complete in {pass_num} passes")
    else:
        self.sorted_pages = []
        print("\nMerge complete (no data)")

    print(f"Total I/O operations: {self.io_count}")

    # Display final sorted result
    print("\nFinal sorted pages:")
    for i, page in enumerate(self.sorted_pages):
        print(f"Page {i:03d}: {page}")

    # Verify sorting result
    if self.sorted_pages:
        all_items = [item for page in self.sorted_pages for item in page]
        is_sorted = all(all_items[i] <= all_items[i+1] for i in
range(len(all_items)-1))

        print(f"\nVerification: Data is {'sorted' if is_sorted else 'NOT sorted'}")
        print(f"Total pages: {len(self.sorted_pages)}")
        print(f"Total items: {len(all_items)}")
    else:
        print("\nVerification: No data to verify")
    return self.sorted_pages

# Main execution
print("\n===== EXTERNAL MERGE SORT =====")
ems = ExternalMergeSort(total_pages=100, page_size=4, buffer_size=3)
ems.generate_data()
ems.sort_phase()

# Get final sorted page list
sorted_pages = ems.merge_phase()

# Display summary view
if sorted_pages:
    all_sorted_items = [item for page in sorted_pages for item in page]
    print(all_sorted_items[:400])
else:
    print("\nNo sorted data available")
```

```
===== EXTERNAL MERGE SORT =====
Generating 100 pages:
Page 00: [147, 262, 549, 799]
Page 01: [78, 483, 816, 920]
Page 02: [290, 458, 736, 757]
Page 03: [175, 335, 397, 654]
Page 04: [66, 772, 798, 915]
Page 05: [20, 536, 731, 919]
Page 06: [4, 114, 218, 786]
Page 07: [153, 553, 631, 861]
Page 08: [199, 405, 485, 636]
Page 09: [119, 130, 220, 751]
Page 10: [232, 471, 564, 796]
Page 11: [101, 425, 846, 981]
Page 12: [11, 66, 907, 977]
Page 13: [368, 549, 940, 972]
Page 14: [325, 559, 603, 784]
Page 15: [30, 84, 378, 738]
Page 16: [150, 407, 429, 826]
Page 17: [198, 334, 673, 768]
Page 18: [164, 262, 696, 725]
Page 19: [114, 525, 576, 664]
Page 20: [124, 497, 623, 798]
Page 21: [142, 276, 302, 605]
Page 22: [271, 453, 498, 944]
Page 23: [59, 618, 824, 828]
Page 24: [390, 880, 903, 988]
Page 25: [60, 155, 678, 947]
Page 26: [210, 243, 766, 882]
Page 27: [95, 466, 553, 927]
Page 28: [231, 235, 351, 615]
Page 29: [157, 435, 561, 576]
Page 30: [286, 828, 851, 911]
Page 31: [83, 213, 479, 938]
Page 32: [508, 688, 808, 920]
Page 33: [633, 644, 809, 963]
Page 34: [400, 434, 440, 978]
Page 35: [363, 401, 447, 667]
Page 36: [77, 318, 347, 576]
Page 37: [20, 61, 331, 632]
Page 38: [461, 517, 770, 852]
Page 39: [166, 393, 477, 631]
Page 40: [10, 281, 847, 853]
Page 41: [153, 272, 989, 996]
Page 42: [12, 43, 178, 442]
Page 43: [64, 412, 727, 805]
Page 44: [53, 118, 236, 920]
Page 45: [440, 471, 605, 853]
Page 46: [46, 356, 441, 873]
Page 47: [254, 522, 699, 714]
Page 48: [332, 773, 828, 962]
Page 49: [67, 491, 747, 758]
Page 50: [392, 417, 724, 849]
Page 51: [518, 558, 939, 994]
Page 52: [557, 636, 904, 935]
Page 53: [4, 77, 487, 791]
Page 54: [333, 593, 763, 850]
Page 55: [317, 343, 856, 904]
Page 56: [130, 396, 730, 915]
Page 57: [151, 459, 643, 795]
Page 58: [557, 715, 767, 995]
Page 59: [47, 237, 367, 544]
Page 60: [66, 368, 410, 413]
Page 61: [148, 249, 308, 392]
```

Page 62: [41, 151, 947, 966]
Page 63: [364, 460, 473, 672]
Page 64: [188, 275, 738, 959]
Page 65: [235, 384, 899, 956]
Page 66: [29, 421, 471, 492]
Page 67: [408, 607, 619, 732]
Page 68: [377, 381, 513, 701]
Page 69: [342, 399, 524, 880]
Page 70: [146, 525, 625, 952]
Page 71: [48, 56, 341, 893]
Page 72: [191, 631, 775, 826]
Page 73: [68, 175, 613, 749]
Page 74: [266, 409, 563, 836]
Page 75: [144, 639, 667, 672]
Page 76: [79, 406, 501, 936]
Page 77: [16, 209, 672, 880]
Page 78: [461, 546, 717, 986]
Page 79: [184, 445, 625, 839]
Page 80: [80, 237, 367, 507]
Page 81: [243, 391, 732, 968]
Page 82: [118, 180, 418, 583]
Page 83: [94, 554, 724, 825]
Page 84: [182, 588, 743, 816]
Page 85: [125, 335, 338, 389]
Page 86: [37, 280, 393, 507]
Page 87: [431, 745, 770, 923]
Page 88: [179, 678, 749, 806]
Page 89: [121, 338, 948, 953]
Page 90: [558, 581, 867, 987]
Page 91: [164, 273, 410, 417]
Page 92: [423, 437, 634, 813]
Page 93: [385, 404, 702, 908]
Page 94: [158, 365, 674, 964]
Page 95: [252, 263, 480, 898]
...
125, 130, 130, 142, 144, 146, 147, 148, 150, 151, 151, 153, 153, 155, 157, 158, 164,
→ 166,
175, 175, 178, 179, 180, 182, 184, 188, 191, 198, 199, 209, 210, 213, 218, 220, 222, 231,
→ 232,
235, 235, 236, 237, 237, 243, 243, 249, 252, 254, 262, 262, 263, 266, 271, 272, 273, 275,
→ 276,
280, 281, 286, 290, 302, 308, 317, 318, 325, 331, 332, 333, 334, 335, 335, 338, 338, 341,
→ 342,
343, 347, 351, 356, 363, 364, 365, 367, 367, 368, 368, 377, 378, 381, 384, 385, 389, 390,
→ 391,
392, 392, 393, 393, 396, 397, 399, 400, 401, 404, 405, 406, 407, 408, 409, 410, 410, 412,
→ 413,
416, 417, 417, 418, 421, 423, 425, 429, 431, 434, 435, 437, 439, 440, 440, 441, 442, 445,
→ 447,
453, 458, 459, 460, 461, 461, 462, 466, 471, 471, 471, 473, 477, 479, 480, 483, 485, 487,
→ 491,
492, 497, 498, 501, 507, 507, 508, 513, 517, 518, 522, 522, 524, 525, 525, 536, 538, 544,
→ 546,
549, 549, 553, 553, 554, 557, 557, 558, 558, 559, 561, 563, 564, 576, 576, 576, 581, 583,
→ 588,
593, 603, 605, 605, 607, 613, 615, 618, 619, 623, 625, 625, 631, 631, 631, 632, 633, 634,
→ 636,
636, 639, 643, 644, 649, 654, 662, 664, 667, 667, 672, 672, 672, 672, 673, 674, 678, 678,
→ 688,
696, 699, 701, 702, 714, 715, 717, 724, 724, 725, 727, 730, 731, 732, 732, 736, 738, 738,
→ 743,
745, 747, 749, 749, 751, 757, 758, 763, 766, 766, 767, 768, 770, 770, 772, 773, 775, 784,
→ 786,
791, 795, 796, 797, 798, 798, 799, 805, 806, 808, 808, 809, 813, 816, 816, 824, 825, 826, 826,
→ 828,

```
828, 828, 833, 836, 839, 846, 847, 849, 850, 851, 852, 853, 853, 856, 861, 863, 867, 873,  
→ 880,  
880, 880, 882, 887, 893, 898, 899, 903, 904, 904, 907, 908, 911, 915, 915, 919, 920, 920,  
→ 920,  
923, 927, 935, 936, 938, 939, 940, 944, 944, 947, 947, 948, 952, 953, 956, 959, 962, 963,  
→ 964,  
966, 968, 972, 977, 978, 981, 986, 987, 988, 989, 994, 995, 996]
```

A rectangular redaction box with a thin black border, spanning most of the width of the page below the header.A second rectangular redaction box with a thin black border, positioned directly below the first one.

7 Advanced Algorithmic Complexity

Advanced Algorithmic Complexity

Date	Author	Comment
2025/08/17 23:12	Yi Guan	Ready to Mark
2025/08/17 23:12	Yi Guan	My handwriting is worse than my drawing on the computer, so I can write out the calculations by hand if necessary.
2025/08/22 20:35	Xiangwen Yang	Good work!
2025/08/22 20:35	Xiangwen Yang	Complete

DEAKIN UNIVERSITY

ADVANCED ALGORITHMS

ONTRACK SUBMISSION

Advanced Algorithmic Complexity

Submitted By:

Yi GUAN

ppv

2025/08/17 23:12

Tutor:

Xiangwen YANG

August 17, 2025



1. Basics of Algorithmic Complexity

Algorithmic complexity measures the time and space resources consumed by an algorithm as the input size grows. Time complexity describes the rate of increase in the number of execution steps and is often expressed in asymptotic notation (O , Ω , Θ); space complexity describes the rate of increase in memory usage.

2. Recurrences

The time complexity of a divide-and-conquer algorithm is typically expressed as the recursive formula: $T(n) = aT(n/b) + f(n)$. Here, a represents the number of subproblems, n/b represents the size of the subproblems ($b > 1$), and $f(n)$ represents the complexity of the split and merge steps.

3. Substitution Method

The substitution method uses mathematical induction to prove a solution by guessing the solution. The steps are: (1) guess the form of the solution ;(2) assume that the solution holds for all sizes smaller than n ; (3) substitute the solution into the recursive form to verify that n holds.

4. Master Theorem

The Master Theorem is used to quickly solve recursive forms of the form $T(n) = aT(n/b) + f(n)$, which can be divided into three cases:

- Case 1 (Leaf-node Dominant): If $f(n) = O(n^{\{log_b a - \varepsilon\}})$ ($\varepsilon > 0$), then $T(n) = \Theta(n^{\{log_b a\}})$.
- Case 2 (balanced at each level): If $f(n)=\Theta(n^{\{log_b a\}} \log^k n)$ ($k \geq 0$), then $T(n)=\Theta(n^{\{log_b a\}} \log^{k+1} n)$.
- Case 3 (root-dominated): If $f(n)=\Omega(n^{\{log_b a+\varepsilon\}})$ ($\varepsilon > 0$) and the regularity condition is met, then $T(n)=\Theta(f(n))$.

6. Selection Algorithm (Select(A, k) Problem)

The selection algorithm is used to find the k th smallest element in an unsorted array, with a worst-case time complexity of $O(n)$.

Task 2: Recurrence using Substitution Method

Solve following recurrence using substitution method:

- $T(n) = \begin{cases} nT(n - 1), & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$
- $T(n) = \begin{cases} T(n - 1) + \log n, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$
- $T(n) = \begin{cases} 3T(n/2) + n, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$

A :

1. $T(n) = n * T(n-1)$, then :

$T(n-1) = (n-1) * T(n-2)$, we can put $T(n-1)$ back in to the first one.

$T(n) = n * (n-1) * T(n-2)$. So we follow this logic we can get rest of the functions:

$$T(n) = n * (n-1) * (n-2) * T(n-3) \dots \dots \dots * T(1)$$

Because when $n=1$ the $T(n) = 1$, therefore that's where the function end.

Finally we can get $T(n) = n * (n-1) * (n-2) * \dots \dots \dots = n!$

$$T(n) = n!$$

2. $T(n)=T(n-1)+\log n$:

$T(n-1) = T(n-2) + \log(n-1)$, we can put $T(n-1)$ back in to the first one.

$T(n) = T(n-2) + \log(n-1) + \log n$, continue:

$$T(n) = T(n-3) + \log(n-2) + \log(n-1) + \log n \dots \dots \dots \text{etc},$$

Because we know that $T(1) = 1$, $\log 1 = 0$

So $T(n) = 1 + \log(n!)$

3. $T(n) = 3T(n/2) + n$

$T(n/2) = 3T(n/4) + n/2$, we can put $T(n/2)$ back in to the first one.

$$T(n) = 3[3T(n/4) + n/2] + n \Rightarrow 9T(n/4) + 3n/2 + n. \text{ Continue:}$$

$$T(n/4) = 3T(n/8) + n/4$$

$$T(n) = 9[3T(n/8) + n/4] + 3n/2 + n \Rightarrow 27T(n/8) + 9n/4 + 3n/2 + n$$

Find out the rule for $F(n)$:

$$T(n) = 3^x * T(n/2^x) + n * \sum k-1/i = 0 (3/2)^i$$

When $n/2^x = 1$, so we will have :

$$T(n) = 3^{\log_2(n)} * T(1) + n * \sum \log_2(n-1)/i = 0 (3/2)^i$$

Because base on the rule for $a^{\log_b(c)} = c^{\log_b(a)}$

So $3^{\log_2(n)} = n^{\log_2(3)}$, plus $\log_2(3) = 1.58$ so is larger than 1.

Therefore, $T(n) = 3n^{\log_2(3)} - 2n$.

$$\boxed{T(N) = n^{\log_2(3)} \times 1 + n \times 2 [n^{\log_2(3-1)} - 1]} \\ \Rightarrow 3n^{\log_2(3)} - 2n$$

Task 3: Recurrence using Master Method

Solve following recurrence using Master Theorem:

$$\bullet \quad T(n) = \begin{cases} T(\sqrt{n}) + \log n, & \text{if } n > 1 \\ 1, & \text{if } n = 1 \end{cases}$$

A :

$$T(n) = T(\sqrt{n}) + \log(n)$$

let $n = 2^m$, so $\sqrt{n} = 2^{m/2}$, $\log n = m$

Let $S(m) = T(2^m)$

$$S(m) = S\left(\frac{m}{2}\right) + m$$

apply Master Method

$$S(m) = aS\left(\frac{m}{b}\right) + f(m), \text{ when } a=1, b=2, f(m)=m, \text{ so}$$

$$m^{\log_b a} \Rightarrow m^{\log_2 1} \Rightarrow m^0 = 1$$

$$S(m) = m + \frac{m}{2} + \frac{m}{4} + \dots \Rightarrow 2m (\Theta(m))$$

Put back To $T(n)$

$$T(n) = S(\log_2(n)) \Rightarrow \Theta(\log(n))$$

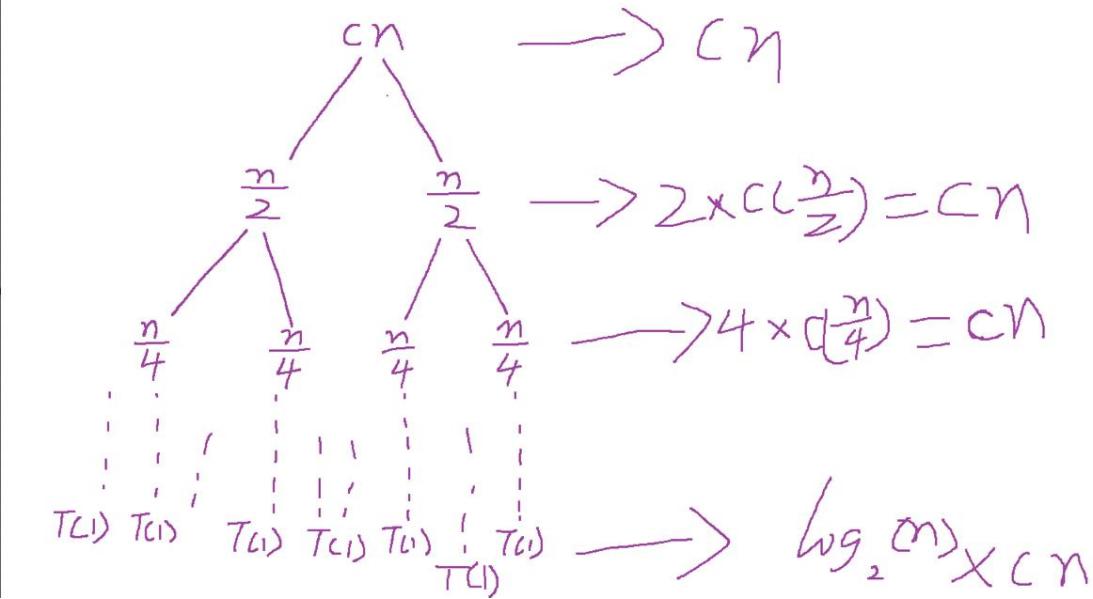
$$\text{so, } T(n) = \Theta(\log(n))$$

Task 4: Recurrence using Recursion Tree Method

Solve following recurrence using recursion tree method:

- $T(n) = 2T(n/2) + cn$

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$



The tree height : $n/2^h = 1$, so $h = \log_2(n)$.

Number of leaf nodes: $2^{\log_2(n)} = n$

Leaf layer cost: $n \cdot T(1) = \theta(n)$, when $T(1) = \theta(1)$

So

$$T(n) = cn \cdot \log_2(n) + \theta(n) = \theta(n \cdot \log(n))$$

Task 5: Solving select(A,K) Recurrence

We had a detailed analysis of $\text{select}(A,k)$ problem in this module. Based on what you learned from this problem, analyse the complexity of Quick Sort Algorithm. Note, quick sort will lead to unbalanced sub-problems (much like $\text{select}(A,k)$) problem. You are expected to use either Akra-Bazzi or induction method to solve the recurrence.

$$T(n) = \frac{2}{n} \sum_{q=0}^{n-1} T(q) + \Theta(n)$$

prove that: $T(n) \leq cn \times \log(n)$

let $k < n$, $T(k) \leq ck \times \log(k)$

$$\begin{aligned} \therefore T(n) &\leq \frac{2}{n} \times \sum_{q=0}^{n-1} [cq \times \log(q)] + D_n \\ &\Rightarrow \frac{2c}{n} \times \sum_{q=2}^{n-1} [q \times \log(q)] + O(1) + D_n \end{aligned}$$

$$\begin{aligned} \therefore \sum_{q=2}^{n-1} q \times \log(q) &\leq \int_2^n x \cdot \log(x) dx \\ &\leq \left[\frac{x^2}{2} \times \log(x) - \frac{x^2}{4} \right]_2^n \\ &\leq \left(\frac{n^2}{2} \times \log(n) - \frac{n^2}{4} \right) \\ \therefore T(n) &\leq \left(\frac{2c}{n} \right) \times \left(\frac{n^2}{2 \log(n)} - \frac{n^2}{4} \right) + D_n \\ &\Rightarrow cn \cdot \log(n) - \frac{cn}{2} + D_n \end{aligned}$$

Therefore, when $c \geq 2D$

$$\begin{aligned} T(n) &\leq cn \cdot \log(n) - \frac{cn}{2} + D_n \\ &\leq cn \cdot \log(n) \end{aligned}$$

Because quick sort selects a pivot each time, divides the array into left and right parts, and sorts recursively.

We assume that the size of the left subarray after partitioning is "q", and "D" is a constant of $\Theta(n)$.

So if a constant number C is larger than 0, therefore $K < n$, $T(k) \leq ck \cdot \log(k)$

If we pick $n=2$:

The $T(2) = 2/2[T(0) + T(1)] + \theta(2) \Rightarrow \theta(1)$

Which means that if $\theta(1) \leq 2c * \log 2$, so $C \geq \theta(1) / 2$

At the end of the function, we need to prove that the

$$cn * \log(n) - cn/2 + Dn + O(1) \leq cn * \log(n)$$

So

$$-cn/2 + Dn + O(1) \leq 0$$

When $C \geq 2D$:

$$n * (D - c/2) \leq 0$$

Therefore : $T(n) \leq cn * \log(n)$

The average time complexity of quick sort is $O(n \log n)$

Reference:

1. <https://blog.csdn.net/yangzhou/article/details/105339108> ;
yangzhou;6/4/2020;
2. <https://www.cnblogs.com/hithongming/p/9200705.html> ; HongmingYou ;
3. https://www.bilibili.com/video/BV1kQ4y1o7jx/?spm_id_from=333.337.search-card.all.click&vd_source=3492fb7dfe3b2952c09c6868f0bcfd4a ; 2021-05-21
04:04:02 ; 雾漫大武汉
4. <https://zhuanlan.zhihu.com/p/267890781> ; 2020-10-24; **王金戈**

8 Graphs II

Taks for Graphs II Module

Date	Author	Comment
2025/08/17 23:10	Yi Guan	Ready to Mark
2025/08/17 23:10	Yi Guan	hihi
2025/08/22 20:38	Xiangwen Yang	Good work!
2025/08/22 20:38	Xiangwen Yang	Complete

DEAKIN UNIVERSITY

ADVANCED ALGORITHMS

ONTRACK SUBMISSION

Graphs II

Submitted By:

Yi GUAN

ppv

2025/08/17 23:10

Tutor:

Xiangwen YANG

August 17, 2025



Reflection :

The shortest path problem finds the minimum distance between two nodes in a graph. Its solution depends on whether the edges are weighted.

For unweighted graphs, the shortest path is measured by the number of edges. Breadth-first search (BFS) solves this problem efficiently, with a running time proportional to the number of nodes and edges.

For weighted graphs, the shortest path is defined as the path with the smallest total weight.

Dijkstra's algorithm finds the shortest path from a source to all other nodes. It runs efficiently even for non-negative weights. Its basic form runs in $O(n^2)$, and can be improved using a heap. Dijkstra's algorithm cannot handle negative weights; any change in edge weights requires rerunning the algorithm.

The Bellman-Ford algorithm can also compute single-source shortest paths, but it can handle negative weights as long as there are no negative cycles. However, because it repeatedly relaxes all edges, it is slower, running in $O(n \times m)$.

Both methods embody the principle of dynamic programming, which solves problems by combining solutions to smaller subproblems.

For the all-pairs shortest path problem, the Floyd-Warshall algorithm is commonly used. This algorithm checks for each pair of nodes whether adding an intermediate node improves the path. Its runtime is $O(n^3)$. It is simpler to implement than running other algorithms multiple times and can handle negative weights, but not negative cycles.

Task 2: Graph Colouring Problem

Implement a greedy algorithm to colour an undirected graph using as few colours as possible.

A simple greedy is — start with an empty colour assignment (e.g., a dictionary where each node will get a colour). Next, for each node in the graph a) look at its neighbours and note which colours have already been used, b) assign the smallest available colour (from 0, 1, 2) that is not used by any neighbour. If a node can't be coloured using any of the 3 allowed colours, mark it as uncolourable (e.g., set to None).

- Assume no more than 3 colours.
- Your algorithm should assign a colour to each node such that no two adjacent nodes have the same colour.
- Discuss whether your approach always produces an optimal colouring. When does it fail?
- Why is your approach greedy?

A :

The general implementation of this method is as follows:

1. First, check which colors are already used by the points next to it.
2. Between 0, 1, and 2, pick the smallest color that its neighbors do not already use and assign it to that point.
3. If its neighbors already use all three colors, mark the point as "uncolored."

Always produces an optimal coloring? When does it fail?

Not always. Changing the order in which points are checked can lead to completely different results. Sometimes, the entire image could be painted with just three colors, but a poor initial choice prevents further painting.

Why "greedy"?

Because each step focuses solely on the immediate present, it chooses the smallest available color for the current point and doesn't care if this will lead to errors later on.

Task 3: Test if Graph is Bi-partite

A graph is bipartite if its nodes can be divided into two disjoint sets such that no two nodes within the same set are adjacent. BFS can be used to check this property by attempting to colour the graph using two colors.

- Design an algorithm that determines whether a given graph is bipartite using BFS. Feel free to modify the BFS code provided in this week's lab notebook.
- Test your implementation on both bipartite and non-bipartite graphs and clearly report your results.

A:

Results

Testing on a bipartite graph:

The algorithm successfully completes the coloring and determines that the graph is bipartite.

Example: The four-node chain structure 1-2-3-4 can be split into two sets: {1,3} and {2,4}.

Testing on a non-bipartite graph:

The algorithm detects a conflict and correctly reports that the graph is not bipartite.

Example: The three-node cycle 1-2-3-1 cannot be split into two sets.

Task 6: Research and Implement Johnson's Algorithm

Research and implement Johnson's algorithm for all-pairs shortest paths, and compare its performance with Floyd-Warshall. Your implementation should use Bellman-Ford as a subroutine.

A:

The Johnson algorithm is an all-source shortest path algorithm suitable for cases with negative-weight edges but no negative-weight cycles. Its main ideas are:

1. Introducing an auxiliary node: Add a new source node q to the graph, connecting it to all other nodes with an edge weight of 0;

2. Running the Bellman-Ford algorithm: Using q as the source, calculate the "potential energy" $h(v)$ of each node, which is used to reweight the edges;

3. Reweighting edges: Modify the weight of each edge (u, v)

$$w'(u, v) = \omega(u, v) + h(u) - h(v)$$

Performance Comparison (with Floyd-Warshall)

Johnson's Algorithm

It has a complexity of approximately $O(V^2 \log V + VE)$ and performs better on sparse graphs ($E \ll V^2$). It uses Dijkstra's algorithm multiple times and is suitable for both sparse and large graphs.

The Floyd-Warshall Algorithm

It has a complexity of $O(V^3)$ and is suitable for small dense graphs. It is simple and straightforward, but less efficient on large graphs.

Resources:

1. <https://medium.com/algorithm-solving/johnsons-algorithm-c461506fccae> ;
Aaron; Apr 26 2020; Johnson's Algorithm
2. <https://blog.csdn.net/myRealization/article/details/124190455> ; memcpy0;
15,04,2022;

1 Lab 6: Graphs II

Lab associated with Module 6: Graphs II

CELL 04

```
# The following lines are used to increase the width of cells to utilize more space on the
# screen
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:95% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

1.0.1 Section 0: Imports

CELL 07

```
import numpy as np
```

CELL 08

```
import math
```

CELL 09

```
from IPython.display import Image
from graphviz import Digraph
import heapq
from collections import deque
from collections import defaultdict
import time
from time import perf_counter
import matplotlib.pyplot as plt
from IPython.display import Image
import random
```

Details of Digraph package: <https://h1ros.github.io/posts/introduction-to-graphviz-in-jupyter-notebook/>

1.0.2 Section 1: Graph Preliminaries

Let us start by implementing our Node and Graph data structure to incorporate weights in the graph

CELL 14

```
class Node:

    def __init__(self, v):

        self.value = v
        self.inNeighbors = []
        self.outNeighbors = []

        #Adrian's suggestion:
        self.parent = None

        self.status = "unvisited"
        self.estD = np.inf

    def hasOutNeighbor(self, v):

        if v in self.outNeighbors:
            return True

        return False

    def hasInNeighbor(self, v):

        if v in self.inNeighbors:
            return True

        return False
```

```
def hasNeighbor(self, v):
    if v in self.inNeighbors or v in self.outNeighbors:
        return True
    return False

def getOutNeighbors(self):
    return self.outNeighbors

def getInNeighbors(self):
    return self.inNeighbors

def getOutNeighborsWithWeights(self):
    return self.outNeighbors

def getInNeighborsWithWeights(self):
    return self.inNeighbors

# -----
# Let us modify following two functions to incorporate weights
# -----


def addOutNeighbor(self,v,wt):
    self.outNeighbors.append((v,wt))

def addInNeighbor(self,v,wt):
    self.inNeighbors.append((v,wt))

def __str__(self):
    return str(self.value)
```

CELL 15

```

class Graph:

    def __init__(self):
        self.vertices = []

    def addVertex(self,n):
        self.vertices.append(n)

    # -----
    # Let us modify following two functions to incorporate weights
    # -----

    def addDiEdge(self, u, v, wt = 1):
        u.addOutNeighbor(v, wt = wt)
        v.addInNeighbor(u, wt = wt)

    # add edges in both directions between u and v
    def addBiEdge(self, u, v, wt = 1):
        self.addDiEdge(u, v, wt = wt)
        self.addDiEdge(v, u, wt = wt)

    # get a list of all the directed edges
    # directed edges are a list of two vertices
    def getDirEdges(self):
        ret = []
        for v in self.vertices:
            ret += [ [v, u] for u in v.outNeighbors ]
        return ret

    # reverse the edge between u and v. Multiple edges are not supported.
    def reverseEdge(self,u,v):
        if u.hasOutNeighbor(v) and v.hasInNeighbor(u):
            if v.hasOutNeighbor(u) and u.hasInNeighbor(v):
                return

            self.addDiEdge(v, u)
            u.outNeighbors.remove(v)
            v.inNeighbors.remove(u)

    def __str__(self):

        ret = "Graph with:\n"
        ret += "\tVertices:\n\t"
        for v in self.vertices:
            ret += str(v) + ","
        ret += "\n"
        ret += "\tEdges:\n\t"
        for a,b in self.getDirEdges():
            ret += "(" + str(a) + "," + str(b) + ") "
        ret += "\n"
        return ret

```

Now that we have incorporated the weights, let us devise a plan to generate the graph

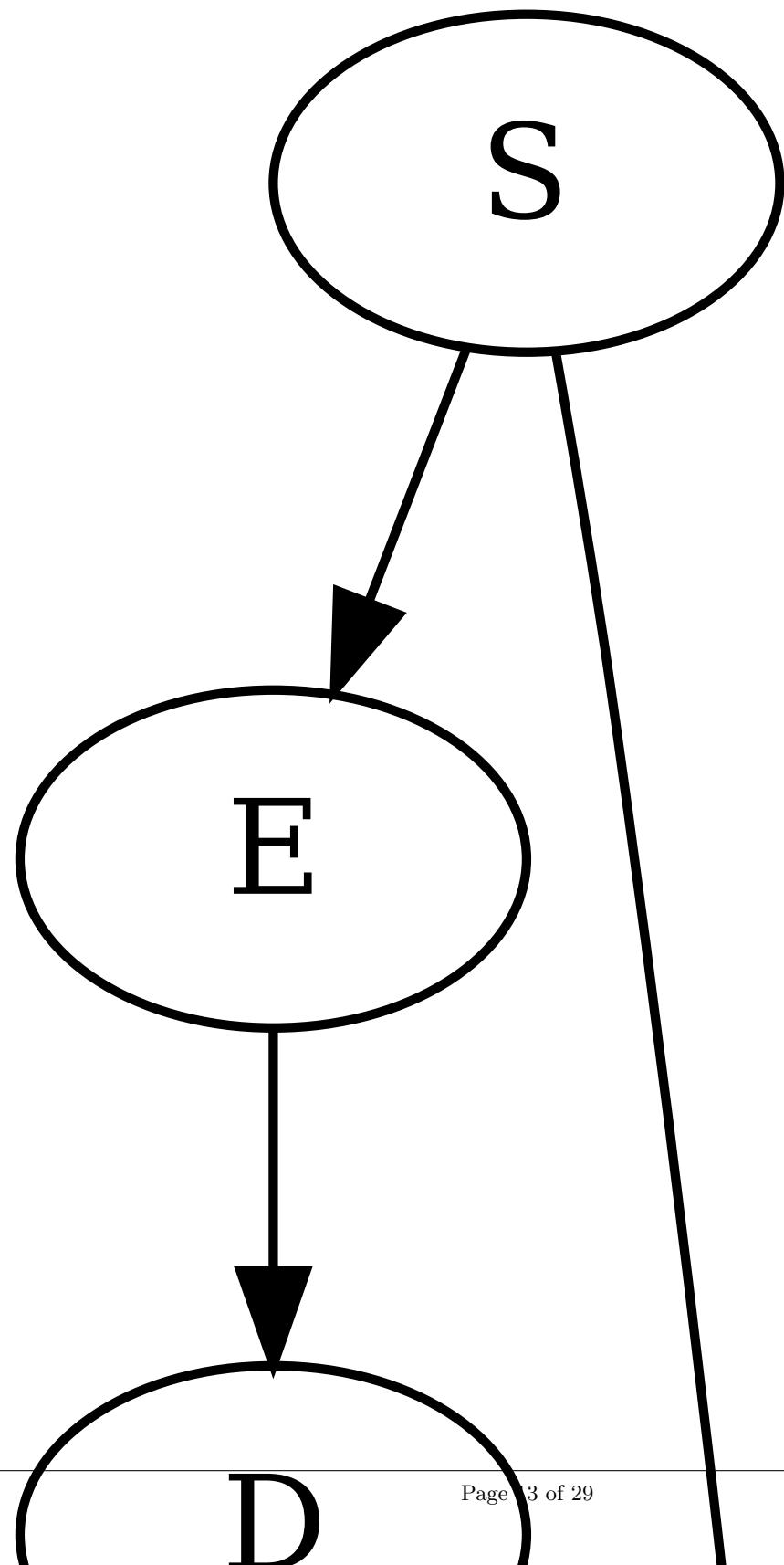
CELL 17

```
dot = Digraph()

dot.edge('S', 'E')
dot.edge('S', 'A')
dot.edge('E', 'D')
dot.edge('D', 'A')
dot.edge('A', 'C')
dot.edge('B', 'A')
dot.edge('D', 'C')
dot.edge('C', 'B')

#dot.view()
dot
```





CELL 18

```
G = Graph()
#for i in ['0', '1', '2', '3', '4', '5']:
for i in ['S', 'E', 'A', 'D', 'B', 'C']:
    G.addVertex( Node(i) )
```

CELL 19

```
V = G.vertices

#dot.edge('S', 'E')
G.addDiEdge( V[0], V[1], 8)

#dot.edge('S', 'A')
G.addDiEdge( V[0], V[2] , 10)

#dot.edge('E', 'D')
G.addDiEdge( V[1], V[3], 1)

#dot.edge('D', 'A')
G.addDiEdge( V[3], V[2], -4)

#dot.edge('A', 'C')
G.addDiEdge( V[2], V[5], 2)

#dot.edge('B', 'A')
G.addDiEdge( V[4], V[2], 1 )

#dot.edge('D', 'C')
G.addDiEdge( V[3], V[5], -1)

#dot.edge('C', 'B')
G.addDiEdge( V[5], V[4], -2 )
```

Second Example

CELL 21

```
G = Graph()
#for i in ['0', '1', '2', '3', '4', '5']:
for i in ['S', 'A', 'B']:
    G.addVertex( Node(i) )

V = G.vertices

G.addDiEdge( V[0], V[1], 3)

G.addDiEdge( V[0], V[2] , 4)

G.addDiEdge( V[2], V[1], -2)
```

CELL 22

```
print(G)
```

```
Graph with:  
Vertices:  
S,A,B,  
Edges:  
(S,<__main__.Node object at 0x0000015501195640>, 3)) (S,<__main__.Node object at  
0x0000015501195790>, 4)) (B,<__main__.Node object at 0x0000015501195640>, -2))
```

1.0.3 Section 2: Dijkstra Algorithm

CELL 25

```
len(G.vertices)
```

3

CELL 26

```

def dijkstra(w, G):

    for v in G.vertices:
        v.estD = math.inf
        v.parent = None

    w.estD = 0
    unsureVertices = G.vertices[:]

    sureVertices = []

    #Nayyar suggestion: parents = [None for i in len(G.vertices)]
    #Adrian's suggestion:
    #for i in len(G.vertices):
    #    G.vertices.index(i).parent = None
    # Jamie's solution: Keep a stack as local variable

    while len(unsureVertices) > 0:

        # find the u with the minimum estD in the dumbest way possible
        u = None
        minD = math.inf
        for x in unsureVertices:
            if x.estD < minD:
                minD = x.estD
                u = x

        if u == None:
            # then there is nothing more that I can reach
            return

        # update u's neighbors
        for v, wt in u.getOutNeighborsWithWeights():

            if v in sureVertices:
                continue

            if u.estD + wt < v.estD:
                v.estD = u.estD + wt

            #Nayyar's solution: parents[G.vertices.index(v)] = G.vertices.index(u)
            #Adrian's solution:
            v.parent = u

        unsureVertices.remove(u)
        sureVertices.append(u)

    # that's it! Now each vertex holds estD which is its distance from w

```

CELL 27

```

w = G.vertices[0]
dijkstra(w, G)

for v in G.vertices:
    print(v.value, v.estD)

```

CELL 28

```
S 0  
A 3  
B 4
```

CELL 29

```
S = w
T = None

for v in G.vertices:
    if v.value == 'B':
        T = v
        break

if T is None:
    print("Nothing")
else:
    current = T
    path = []
    while current != S:
        path.append(current)
        current = current.parent

path.append(S)
path.reverse()
print(f"from {S.value} to {T.value} path:", [node.value for node in path])
print(f"distance: {T.estD}")
```

```
from S to B path: ['S', 'B']
distance: 4
```

CELL 30

1.0.4 Activity 1: Modify above code to do Bellman-Ford. Make sure you test for negative cycles and compare its performance with Dijkstra.

CELL 33

```
#### TODO #####
### Good Luck ###

def Bellman_Ford(w, G):
    # Initialize distance and predecessor for all vertices
    for v in G.vertices:
        v.estD = math.inf
        v.parent = None

    w.estD = 0

    # Collect all directed edges (with weights)
    edges = []
    for v in G.vertices:
        for neighbor, wt in v.getOutNeighborsWithWeights():
            edges.append((v, neighbor, wt))

    # Perform |V|-1 relaxation steps
    for i in range(len(G.vertices) - 1):
        updated = False
        for u, v, wt in edges:
            if u.estD != math.inf and u.estD + wt < v.estD:
                v.estD = u.estD + wt
                v.parent = u
                updated = True

        if not updated:
            break

    # Check for negative-weight cycle
    has_negative_cycle = False
    for u, v, wt in edges:
        if u.estD != math.inf and u.estD + wt < v.estD:
            has_negative_cycle = True
            print("Negative cycle detected!")
            break

    return has_negative_cycle

def reconstruct_path(start, end):
    """Reconstruct path from end node to start node"""
    if end.estD == math.inf:
        return [] # Unreachable

    path = []
    current = end
    while current is not None:
        path.append(current)
        # Prevent infinite loop (safety check)
        if current.parent is None or current.parent in path:
            break
        current = current.parent

    path.reverse()
```

```
# Ensure the path starts with the start node
if path and path[0] != start:
    return []
return path

# Test Bellman-Ford
print("Activity 1")

# Create Section 1 graph (6 nodes)
G_sec1 = Graph()
for i in ['S', 'E', 'A', 'D', 'B', 'C']:
    G_sec1.addVertex(Node(i))

V = G_sec1.vertices
G_sec1.addDiEdge(V[0], V[1], 8)    # S -> E
G_sec1.addDiEdge(V[0], V[2], 10)   # S -> A
G_sec1.addDiEdge(V[1], V[3], 1)    # E -> D
G_sec1.addDiEdge(V[3], V[2], -4)   # D -> A
G_sec1.addDiEdge(V[2], V[5], 2)    # A -> C
G_sec1.addDiEdge(V[4], V[2], 1)    # B -> A
G_sec1.addDiEdge(V[3], V[5], -1)   # D -> C
G_sec1.addDiEdge(V[5], V[4], -2)   # C -> B

has_negative_cycle = Bellman_Ford(V[0], G_sec1) # Start from S
print("\nNegative cycle exists?")
print("")
print(has_negative_cycle)
print("")
print("Shortest path distances:")
for v in G_sec1.vertices:
    print(f"{v.value}: {v.estD}")

# S -> B
path = reconstruct_path(V[0], V[4])  # S -> B
print("\nS->B Path:", [node.value for node in path])

# S -> C
path = reconstruct_path(V[0], V[5])  # S -> C
print("S->C Path:", [node.value for node in path])

# S -> A
path = reconstruct_path(V[0], V[2])  # S -> A
print("S->A Path:", [node.value for node in path])
```

```
Activity 1
```

```
Negative cycle exists?
```

```
False
```

```
Shortest path distances:
```

```
S: 0
```

```
E: 8
```

```
A: 5
```

```
D: 9
```

```
B: 5
```

```
C: 7
```

```
S->B Path: ['S', 'E', 'D', 'A', 'C', 'B']
```

```
S->C Path: ['S', 'E', 'D', 'A', 'C']
```

```
S->A Path: ['S', 'E', 'D', 'A']
```

1.0.5 Activity 2: Implement Floyd-Warshall algorithm using above data structure (that is Node and Graph). Make sure to test all the use-cases.

CELL 36

```
#### TODO #####
### Good Luck ###

def Floyd_Warshall(G):
    # Distance and predecessor matrices
    vertices = G.vertices
    n = len(vertices)

    # Create the distance matrix
    distance = np.full((n, n), np.inf)

    # Set diagonal
    for i in range(n):
        distance[i][i] = 0

    # Create the predecessor matrix
    next_node = np.full((n, n), -1) # Stores the index of the next hop

    # Create a mapping
    index_map = {v: i for i, v in enumerate(vertices)}

    for u in vertices:
        u_idx = index_map[u]
        for v, wt in u.getOutNeighborsWithWeights():
            v_idx = index_map[v]
            distance[u_idx][v_idx] = wt
            next_node[u_idx][v_idx] = v_idx # Next hop is the destination node

    for k in range(n):
        for i in range(n):
            if distance[i][k] == np.inf:
                continue
            for j in range(n):
                if distance[k][j] == np.inf:
                    continue

                new_distance = distance[i][k] + distance[k][j]
                if new_distance < distance[i][j]:
                    distance[i][j] = new_distance
                    next_node[i][j] = next_node[i][k] # Next hop points to the
intermediate node

    # Negative cycles
    has_negative_cycle = False
    for i in range(n):
        if distance[i][i] < 0:
            has_negative_cycle = True
            print(f"Negative cycle detected: self-loop distance at node
{vertices[i].value} is {distance[i][i]}")
            break

    # Path reconstruction
    def reconstruct_path(start, end):
        start_idx = index_map[start]
        end_idx = index_map[end]

        if distance[start_idx][end_idx] == np.inf:
            return []

        path = [vertices[end].value]
        current_node = end_idx
        while current_node != start_idx:
            current_node = next_node[current_node]
            path.append(vertices[current_node].value)
        path.append(vertices[start].value)
        path.reverse()
        return path
```

```
path = [start]
current_idx = start_idx

# Reconstruct the path
while current_idx != end_idx:
    next_idx_val = next_node[current_idx][end_idx]
    if next_idx_val == -1:
        break
    path.append(vertices[next_idx_val])
    current_idx = next_idx_val

return path

return distance, next_node, has_negative_cycle, reconstruct_path

# Test
print("Activity 2")

# Use the graph from Section 1
dist_matrix, next_node, has_neg_cycle, reconstruct_path = Floyd_Warshal(G_sec1)
print("\nDistance matrix:")
print(Floyd_Warshal(G_sec1))
print(dist_matrix)

print("\nNegative cycle exists?")
print()
print(has_neg_cycle)

graph_vertices = G_sec1.vertices

node_dict = {node.value: node for node in graph_vertices}

# S -> B
path = reconstruct_path(node_dict['S'], node_dict['B'])
print("\nPath S->B:", [node.value for node in path])

# S -> C
path = reconstruct_path(node_dict['S'], node_dict['C'])
print("Path S->C:", [node.value for node in path])

# S -> A
path = reconstruct_path(node_dict['S'], node_dict['A'])
print("Path S->A:", [node.value for node in path])

# S -> D
path = reconstruct_path(node_dict['S'], node_dict['D'])
print("Path S->D:", [node.value for node in path])
```

Activity 2

```
Distance matrix:  
(array([[ 0.,  8.,  5.,  9.,  5.,  7.],  
       [inf,  0., -3.,  1., -3., -1.],  
       [inf, inf,  0., inf,  0.,  2.],  
       [inf, inf, -4.,  0., -4., -2.],  
       [inf, inf,  1., inf,  0.,  3.],  
       [inf, inf, -1., inf, -2.,  0.]]), array([-1,  1,  1,  1,  1,  1],  
[-1, -1,  3,  3,  3,  3],  
[-1, -1, -1, -1,  5,  5],  
[-1, -1,  2, -1,  2,  2],  
[-1, -1,  2, -1, -1,  2],  
[-1, -1,  4, -1,  4, -1])), False, <function Floyd_Warshal.<locals>.reconstruct_path at  
0x0000015501F7DB80>)  
[[ 0.  8.  5.  9.  5.  7.]  
[inf  0. -3.  1. -3. -1.]  
[inf inf  0. inf  0.  2.]  
[inf inf -4.  0. -4. -2.]  
[inf inf  1. inf  0.  3.]  
[inf inf -1. inf -2.  0.]]  
  
Negative cycle exists?  
  
False  
  
Path S->B: ['S', 'E', 'D', 'A', 'C', 'B']  
Path S->C: ['S', 'E', 'D', 'A', 'C']  
Path S->A: ['S', 'E', 'D', 'A']  
Path S->D: ['S', 'E', 'D']
```

2 Task 2 Graph Colouring Problem

CELL 40

```
def greedy_graph_coloring(graph):
    colors = {}
    for node in graph:
        # Get the set of colours used by neighbours
        neighbour_colors = {colors[neigh] for neigh in graph[node] if neigh in colors}
        # Try assigning a colour (0,1,2)
        for c in range(3):
            if c not in neighbour_colors:
                colors[node] = c
                break
        else:
            # No colour available
            colors[node] = None
    return colors

# Example usage
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'C'],
    'C': ['A', 'B', 'D'],
    'D': ['C']
}
print("Greedy colouring:", greedy_graph_coloring(graph))
```

```
Greedy colouring: {'A': 0, 'B': 1, 'C': 2, 'D': 0}
```

3 Task 3 Test if Graph is Bi-parBte

CELL 42

```
from collections import deque

def is_bipartite(graph):
    color = {}
    for start in graph:
        if start not in color: # No color
            queue = deque([start])
            color[start] = 0
        while queue:
            node = queue.popleft()
            for neigh in graph[node]:
                if neigh not in color:
                    color[neigh] = 1 - color[node] # Mix colour
                    queue.append(neigh)
                elif color[neigh] == color[node]:
                    return False, color
    return True, color

# test
bipartite_graph = {
    1: [2, 3],
    2: [1, 4],
    3: [1, 4],
    4: [2, 3]
}
non_bipartite_graph = {
    1: [2, 3],
    2: [1, 3],
    3: [1, 2]
}
print("Bipartite test 1:", is_bipartite(bipartite_graph))
print("Bipartite test 2:", is_bipartite(non_bipartite_graph))
```

```
Bipartite test 1: (True, {1: 0, 2: 1, 3: 1, 4: 0})
Bipartite test 2: (False, {1: 0, 2: 1, 3: 1})
```

4 Task 6 Task 6: Research and Implement Johnson's Algorithm

CELL 44

```

def dijkstra_adj(graph, start):
    "using adjacency list representation."
    dist = {node: float('inf') for node in graph}
    dist[start] = 0
    heap = [(0, start)]
    while heap:
        d, u = heapq.heappop(heap)
        if d != dist[u]:
            continue
        for v, w in graph[u].items():
            nd = d + w
            if nd < dist[v]:
                dist[v] = nd
                heapq.heappush(heap, (nd, v))
    return dist

def johnson(vertices, edges):
    # Create Graph
    G_bf = Graph()
    node_map = {}

    # Add original vertices
    for v in vertices:
        node = Node(v)
        G_bf.addVertex(node)
        node_map[v] = node

    # Add auxiliary node 'Q'
    node_q = Node('Q')
    G_bf.addVertex(node_q)
    node_map['Q'] = node_q

    # Add original edges
    for u, v, w in edges:
        G_bf.addDiEdge(node_map[u], node_map[v], w)

    # Add edges from 'Q'
    for v in vertices:
        G_bf.addDiEdge(node_q, node_map[v], 0)

    # Bellman-Ford
    has_neg_cycle = Bellman_Ford(node_q, G_bf)
    if has_neg_cycle:
        print("Negative cycle detected. Johnson's algorithm not applicable.")
        return None

    # Get reweighting values
    h = {v: node_map[v].estD for v in vertices}

    # Build list graph
    graph_rew = {v: {} for v in vertices}
    for u, v, w in edges:
        graph_rew[u][v] = w + h[u] - h[v]  # Reweight edges

    # for each vertex and adjust distances
    result = {}
    for u in vertices:
        dists = dijkstra_adj(graph_rew, u)
        result[u] = {}
        for v in dists:
            result[u][v] = dists[v] - h[u] + h[v]  # Original distance

```

```
    return result

# Test out
V = ['A', 'B', 'C', 'D']
E = [(A, B, 1), (B, C, 3), (A, C, -2), (C, D, 2)]
print("Johnson result:", johnson(V, E))
```

```
Johnson result: {'A': {'A': 0, 'B': 1, 'C': -2, 'D': 0}, 'B': {'A': inf, 'B': 0, 'C': 3, 'D': 5}, 'C': {'A': inf, 'B': inf, 'C': 0, 'D': 2}, 'D': {'A': inf, 'B': inf, 'C': inf, 'D': 0}}
```

CELL 45

9 Dynamic Programming

Task associated with Dynamic Programming module

Date	Author	Comment
2025/08/24 21:39	Yi Guan	Ready to Mark
2025/08/24 21:39	Yi Guan	I haven't noticed the deadline for this task discussion; I thought it would be next week. I have sent an email to the uni chair about my situation, hope it's all right. :(
2025/08/29 20:33	Xiangwen Yang	Good work!
2025/08/29 20:33	Xiangwen Yang	Complete

DEAKIN UNIVERSITY

ADVANCED ALGORITHMS

ONTRACK SUBMISSION

Dynamic Programming

Submitted By:

Yi GUAN

ppv

2025/08/24 21:39

Tutor:

Xiangwen YANG

August 24, 2025



Reflection

Introduction to Dynamic Programming

Dynamic Programming (DP) is an effective problem-solving paradigm designed to optimize solutions for complex problems. It works by breaking a problem into smaller overlapping subproblems, storing their results, and building the global optimal solution from them. Unlike divide-and-conquer, DP specifically handles overlapping subproblems and optimal substructure. Through approaches such as top-down memorization and bottom-up tabulation, DP reduces exponential complexity into polynomial time.

Longest Common Subsequence (LCS)

The LCS problem aims to find the longest subsequence common to two given sequences, where order matters but continuity is not required. For example, between “ABAZDC” and “BACBAD,” the LCS is “ABAD.” The DP solution constructs a two-dimensional table to record the length of LCS for all prefix pairs. The final result is obtained by filling this table and backtracking. The time and space complexity are both $O(mn)$, where m and n are the string lengths.

Knapsack Problem

The Knapsack Problem is a classic optimization problem. In the 0/1 Knapsack, each item has a weight and value, and the goal is to maximize the total value without exceeding capacity, where each item can be chosen at most once.

Unbounded Knapsack

In the Unbounded Knapsack, each item can be chosen multiple times. The DP state can be reduced to a one-dimensional array, where $dp[w]$ stores the best value at capacity w .

0/1 Knapsack (Revisited)

Unlike the unbounded version, the 0/1 Knapsack restricts each item to be used once. The DP transition is similar but carefully ensures that items are only considered once per capacity. Space optimization techniques can further reduce memory usage, showing the flexibility of DP approaches.

Task 2: Combinatorial DP

You are climbing a staircase with a total of n steps. At each move, you can hop 1, 2, or 3 steps.

- Write a recursive solution to calculate the number of distinct ways to reach the top.
- Then refactor your solution into a top-down dynamic programming approach using memoization.
- Finally, implement a bottom-up DP solution for comparison.
- Test your implementation for various values of n , and report the time or memory performance (optional but encouraged).

The stair climbing problem requires computing the number of different ways to climb n stairs, taking 1, 2, or 3 steps at a time.

The code implements only a recursive solution, resulting in a time complexity of $O(3^n)$ and a space complexity of $O(n)$.

In contrast, using memorization or bottom-up dynamic programming, the time complexity can be optimized to $O(n)$, while the space complexity remains $O(n)$. Even for $n=1000$, the problem can be completed in milliseconds, demonstrating the significant performance boost that dynamic programming can bring to algorithms.

Task 5: Fewest Coins

Given coins of different denominations and a total amount, find the minimum number of coins needed to make up that amount. Your task is to implement both:

- Top-down memoized solution,
- Bottom-up tabulation version.
- Ensure your solution handles cases where it's not possible to reach the amount.

Example:

Input: coins = [1, 2, 5], amount = 11

Output: 3

Explanation: $11 = 5 + 5 + 1$

Input: coins = [2], amount = 3

Output: -1

Explanation: No combination of coins adds to 3.

Make sure you are ready to get your strategy discussed with your tutor.

The minimum coin problem requires finding a target amount using coins of given denominations while using the minimum number of coins.

The code provides a bottom-up solution, creating a dp array $dp[i]$ representing the minimum number of coins required to produce amount i , starting with the simplest case and building up the solution step by step.

For example, with coins = [1, 2, 5] and amount = 11, the calculation shows that 3 coins are ultimately required ($5 + 5 + 1$).

This method has a time complexity of $O(\text{amount} \times \text{coins_count})$, making it much more efficient than a recursive approach.

The top-down (memoization) approach starts with the target problem, recursively decomposes it into subproblems, and uses memoization to store already computed

results. Its advantages are that it is more intuitive and only computes the necessary subproblems. However, its disadvantage is that it incurs recursive overhead.

The bottom-up (tabulation) approach starts with the simplest subproblem and gradually builds toward the target problem. Its advantages are that it has no recursive overhead and is generally more efficient. However, its disadvantage is that it may compute unnecessary subproblems.

Both approaches guarantee an optimal solution, but the bottom-up approach is generally preferred in practical applications because it avoids the performance overhead of recursive calls and the risk of stack overflow, making it more stable and reliable when handling large-scale problems.

1 Lab 7: Dynamic Programming

Lab associated with Module 7: Dynamic Programming

CELL 04

```
# The following lines are used to increase the width of cells to utilize more space on the
# screen
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:95% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

1.0.1 Section 0: Imports

CELL 07

```
import numpy as np
```

CELL 08

```
import math
```

CELL 09

```
from IPython.display import Image
from graphviz import Digraph
```

Details of Digraph package: <https://h1ros.github.io/posts/introduction-to-graphviz-in-jupyter-notebook/>

1.0.2 Activity 1: You are running up a staircase with a total of n steps. You can hop either 1 step, 2 steps or 3 steps at a time. Write a DP program to determine how many possible ways you can run up the stairs? (Hint: Start with a recursive solution, and then later move to top-down approach of DP).

CELL 13

```
### TODO ###
### Good Luck ####
def count_ways(n):
    if n < 0:
        return 0
    if n == 0:
        return 1
    return count_ways(n-1) + count_ways(n-2) + count_ways(n-3)

for n in [0,1,2,3,4,5,10]:
    print("n =", n, "ways =", count_ways(n))
```

```
n = 0 ways = 1
n = 1 ways = 1
n = 2 ways = 2
n = 3 ways = 4
n = 4 ways = 7
n = 5 ways = 13
n = 10 ways = 274
```

1.0.3 Activity 2: Write the code for finding the Longest Common Sub-sequence. Make sure you output the Matrix C and teh longest sub-sequence as well. Test your code with various use-cases.

CELL 15

```
### TODO ###
### Good Luck ###

def lcs(X, Y):
    m, n = len(X), len(Y)
    dp = [[0] * (n+1) for _ in range(m+1)]

    for i in range(1, m+1):
        for j in range(1, n+1):
            if X[i-1] == Y[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    res = ""
    i, j = m, n
    while i > 0 and j > 0:
        if X[i-1] == Y[j-1]:
            res = X[i-1] + res
            i -= 1
            j -= 1
        elif dp[i-1][j] >= dp[i][j-1]:
            i -= 1
        else:
            j -= 1
    return res, dp
```

CELL 16

```
if __name__ == "__main__":
    X, Y = "ABCBDAB", "BDCABA"
    lcs_str, dp_table = lcs(X, Y)
    print(f"X={X}, Y={Y} -> LCS={lcs_str}")
    print("DP Table:")
    for row in dp_table:
        print(row)
```

```
X='ABCBDBAB', Y='BDCABA' -> LCS='BCBA'  
DP Table:  
[0, 0, 0, 0, 0, 0, 0]  
[0, 0, 0, 0, 1, 1, 1]  
[0, 1, 1, 1, 1, 2, 2]  
[0, 1, 1, 2, 2, 2, 2]  
[0, 1, 1, 2, 2, 3, 3]  
[0, 1, 2, 2, 2, 3, 3]  
[0, 1, 2, 2, 3, 3, 4]  
[0, 1, 2, 2, 3, 4, 4]
```

1.0.4 Section 2: Unbounded Knapsack Problem

Let us build a solution to unbounded Knapsack problem.

CELL 20

```
def unboundedKnapsack(W, n, wt, vals, names):  
  
    K = [0 for i in range(W + 1)]  
    ITEMS = [[] for i in range(W + 1)]  
  
    for x in range(1, W + 1):  
        K[x] = 0  
        for i in range(1, n):  
  
            prev_k = K[x]  
  
            if (wt[i] <= x):  
                K[x] = max(K[x], K[x - wt[i]] + vals[i])  
  
            if K[x] != prev_k:  
                ITEMS[x] = ITEMS[x - wt[i]] + names[i]  
  
    return K[W], ITEMS[W]
```

CELL 21

```
W = 4  
wt = [1, 2, 3]  
vals = [1, 4, 6]  
names = [[ "Turtle" ], [ "Globe" ], [ "WaterMelon" ]]  
  
n = len(vals)  
  
print('We have {} items'.format(n))
```

We have 3 items

CELL 22

```
K, ITEMS = unboundedKnapsack(W, n, wt, vals, names)
```

CELL 23

```
ITEMS
```

```
['Globe', 'Globe']
```

CELL 24

1.0.5 Activity 3: In the earlier activity, you analysed the code for unbounded knapsack. Based on the algorithm discussed in this section, implement a solution to do 0/1 Knapsack. Make sure you test your algorithms for various test-cases.

CELL 27

```
#### TODO ####
### Good Luck ###
def knapsack_01(W, wt, vals, names):
    n = len(vals)
    dp = [[0] * (W+1) for _ in range(n+1)]
    keep = [[[0] for _ in range(W+1)] for _ in range(n+1)]

    for i in range(1, n+1):
        for w in range(1, W+1):
            if wt[i-1] <= w:
                if vals[i-1] + dp[i-1][w - wt[i-1]] > dp[i-1][w]:
                    dp[i][w] = vals[i-1] + dp[i-1][w - wt[i-1]]
                    keep[i][w] = keep[i-1][w - wt[i-1]] + [names[i-1]]
                else:
                    dp[i][w] = dp[i-1][w]
                    keep[i][w] = keep[i-1][w]
            else:
                dp[i][w] = dp[i-1][w]
                keep[i][w] = keep[i-1][w]
    return dp[n][W], keep[n][W]
```

Class Room Test-case

CELL 29

```
W = 10
V = [20, 8, 14, 13, 35]
w = [6, 2, 4, 3, 11]

n = len(vals)

valid_indices = [i for i in range(len(wt)) if wt[i] <= W]
vals_valid = [vals[i] for i in valid_indices]

wt_valid = [wt[i] for i in valid_indices]
names_valid = [names[i] for i in valid_indices]
max_val, items = knapsack_01(W, wt_valid, vals_valid, names_valid)

print(f"Max Value: {max_val}, Items: {items}")
print('We have {} items'.format(n))
```

```
Max Value: 35, Items: ['Item2', 'Item3', 'Item4']
We have 5 items
```

CELL 30

2 Task 5

CELL 33

```
def fewest_coins(coins, amount):
    dp = [float('inf')] * (amount+1)
    dp[0] = 0
    for i in range(1, amount+1):
        for c in coins:
            if i - c >= 0:
                dp[i] = min(dp[i], dp[i-c] + 1)
    return dp[amount] if dp[amount] != float('inf') else -1
```

CELL 34

```
if __name__ == "__main__":
    test_cases = [[(1, 2, 5), 11], ([2], 3)]
    for coins, amount in test_cases:
        result = fewest_coins(coins, amount)
        print(f"Coins: {coins}, Amount: {amount} -> Min Coins: {result}")
```

```
Coins: [1, 2, 5], Amount: 11 -> Min Coins: 3
Coins: [2], Amount: 3 -> Min Coins: -1
```

CELL 35

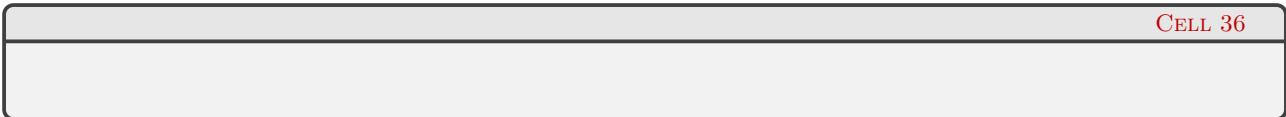
```
def fewest_coins_top_down(coins, amount, memo=None):
    if memo is None:
        memo = {}
    if amount in memo:
        return memo[amount]
    if amount == 0:
        return 0
    if amount < 0:
        return float('inf')

    min_coins = float('inf')
    for coin in coins:
        result = fewest_coins_top_down(coins, amount - coin, memo)
        if result != float('inf'):
            min_coins = min(min_coins, result + 1)

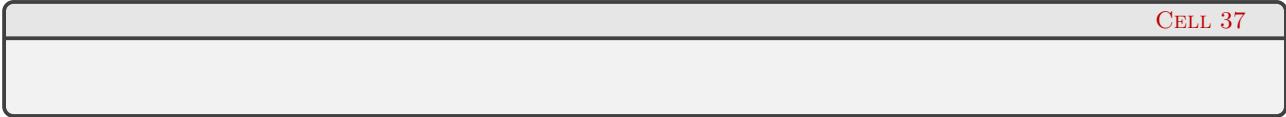
    memo[amount] = min_coins
    return min_coins

result = fewest_coins_top_down([1,2,5], 11)
print(result if result != float('inf') else -1)
```

3



CELL 36



CELL 37

10 Greedy Algorithms

Tasks associated with greedy algorithms module

Date	Author	Comment
2025/08/29 19:43	Yi Guan	Ready to Mark
2025/08/29 19:43	Yi Guan	hihi
2025/08/29 20:38	Xiangwen Yang	Good work!
2025/08/29 20:38	Xiangwen Yang	Complete

DEAKIN UNIVERSITY

ADVANCED ALGORITHMS

ONTRACK SUBMISSION

Greedy Algorithms

Submitted By:

Yi GUAN

ppv

2025/08/29 19:43

Tutor:

Xiangwen YANG

August 29, 2025



Reflection

Greedy algorithms are a fundamental problem-solving paradigm in computer science, characterized by making locally optimal decisions at each step in the hope of reaching a globally optimal solution.

1. Activity Selection

The classic activity selection problem involves selecting the maximum number of non-overlapping activities, each with a start and finish time. By ordering activities by completion time and always selecting the one that finishes first, the algorithm is simple (due to the ordering, the complexity is $O(n \log n)$) and provably optimal. Proof by contradiction confirms that this greedy strategy does not sacrifice global optimality.

2. Job Scheduling

While the image includes "job scheduling," a general greedy approach to solving these problems typically involves ordering jobs by deadline or profit. The system selects the most profitable task, with the goal of maximizing throughput or minimizing latency. This illustrates how greedy techniques can be effective for real-world scheduling problems, although the formal correctness may be more subtle.

3. Huffman Coding

Huffman coding is a greedy algorithm used for lossless data compression. At each step, the two least frequent characters are merged to construct a prefix-free binary tree, thus minimizing the total encoding length. This method ensures optimality for variable-length encoding and is the basis of compression techniques.

4. Minimum Spanning Tree (Prim's Algorithm and Kruskal's Algorithm)

The greedy approach is also the basis of Prim's and Kruskal's algorithms. Both algorithms construct minimum spanning trees on weighted undirected graphs by carefully selecting the edges with the lowest weights. Each step makes the best local choice, either starting from a single node or adding the minimum number of edges between components, and both algorithms obtain the best overall result.

Task 2: Optimised Algorithm for Minimum Spanning Tree

Rewrite Prim's algorithm (code given to you) using a more efficient approach inspired by the Dijkstra-like implementation discussed in the seminar:

- Use a priority queue (e.g., heapq) to select the next edge with the minimum weight and maintain a visited set and a distance/weight map to track eligible edges.
- Your implementation should:
 - Work on undirected, weighted graphs.
 - Efficiently compute the MST with improved runtime.
- Compare the performance (e.g., via timing or number of comparisons) between the un-optimized and optimized versions on a large random graph.

The original Prim's algorithm has a time complexity of $O(V^2)$, where V represents the number of vertices. This quadratic complexity means that the runtime increases quadratically as the graph size increases.

The optimized version reduces the time complexity to $O(E \log V)$, where E represents the number of edges, by introducing a priority queue (min heap). The main cost of the optimized algorithm is increased space complexity, as a priority queue must be maintained.

The main reason for this performance difference is that the original algorithm must traverse all edges of all visited vertices each time it selects the minimum edge. The optimized algorithm, using a priority queue, can retrieve the minimum-weight edge in logarithmic time.

As the graph size increases, the cost of the original algorithm's inner loop increases dramatically, but the optimized algorithm effectively controls this cost through its choice of data structure.

Task 3: Greedy Coin Change Algorithm

Write a Python function that: takes two inputs — a list of positive integers representing coin denominations (e.g., [1, 5, 10, 25]), an integer amount representing the total value to be achieved. And, outputs a list of coins used to make up the amount, *selected using a greedy strategy (i.e., always take the largest coin value that fits into the remaining amount)*. If no combination of coins can make up the amount, return -1.

Compare your results with Dynamic Programming solution. Critically evaluate when this approach works and when it fails.

The greedy algorithm uses an intuitive strategy, always choosing the coin with the largest denomination, to produce the optimal solution. The greedy algorithm has a linear time complexity of $O(n)$ and a constant space complexity of $O(1)$, making it extremely efficient.

However, the greedy algorithm does not always yield the optimal solution. When the coin system does not meet the "standard coin system" condition, that is, when there is no integer multiple relationship between the coin denominations, the greedy algorithm may fail.

For example, in the coin system [1, 5, 10, 21, 25], to make a change of 63 yuan, the greedy algorithm will produce a solution with 5 coins, while the dynamic programming algorithm will find the optimal solution with only 3 21 yuan coins.

The dynamic programming algorithm systematically considers all possible combinations to ensure that the solution with the least number of coins is found. Although dynamic programming can always yield the optimal solution, the computational cost is relatively high, especially for large amounts.

1 Greedy Algorithms

Lab associated with Module 8: Greedy Algorithms

CELL 04

```
# The following lines are used to increase the width of cells to utilize more space on the
# screen
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:95% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

1.0.1 Section 0: Imports

CELL 07

```
import numpy as np
```

CELL 08

```
import math
```

CELL 09

```
from IPython.display import Image
from graphviz import Digraph
import heapq
import time
import random
```

1.0.2 Activity 1: Write code for creating a prefix tree for any arbitrary distribution, e.g., [A:45, B:13, C:12, D:16, E:9, F:5]. Your algorithm should return the prefix tree and should display the correct code for every alphabet..

CELL 12

```
#### TODO ####
### Good Luck ###

class Node:
    def __init__(self, char=None, freq=0):
        self.char = char
        self.freq = freq
        self.left = None
        self.right = None

    def __lt__(self, other): # Sort by frequency when comparing heaps
        return self.freq < other.freq

# Constructing a Huffman tree
def build_tree(freq_dict):
    # 1. Put all characters into the root pile
    heap = [Node(ch, f) for ch, f in freq_dict.items()]
    heapq.heapify(heap)

    # 2. Take the two smallest ones each time
    while len(heap) > 1:
        n1 = heapq.heappop(heap)
        n2 = heapq.heappop(heap)
        parent = Node(freq=n1.freq + n2.freq)
        parent.left = n1
        parent.right = n2
        heapq.heappush(heap, parent)

    return heap[0]

# Traversing the tree
def get_codes(node, code="", code_dict=None):
    if code_dict is None:
        code_dict = {}

    if node.left is None and node.right is None:
        code_dict[node.char] = code
    else:
        get_codes(node.left, code+ "0", code_dict)
        get_codes(node.right, code+ "1", code_dict)

    return code_dict
```

```
if node:  
    if node.char is not None: # Leaf nodes  
        code_dict[node.char] = code  
        get_codes(node.left, code + "0", code_dict)  
        get_codes(node.right, code + "1", code_dict)  
return code_dict
```

CELL 13

```
example = {'A':45, 'B':13, 'C':12, 'D':16, 'E':9, 'F':5}
root = build_tree(example)
codes = get_codes(root)

print("Huffman ")
for ch, code in codes.items():
    print(ch, ":", code)
```

```
Huffman
A : 0
C : 100
B : 101
F : 1100
E : 1101
D : 111
```

1.0.3 Prim's Algorithm

Graph's Preliminaries

```
from graph import *
```

CELL 17

```
G = Graph()  
  
for i in ['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I']:  
    G.addVertex( Node(i) )
```

CELL 18

```
V = G.vertices  
  
#0, 1, 2, 3, 4, 5, 6, 7, 8  
#A, B, C, D, E, F, G, H, I  
  
G.addBiEdge( V[0], V[1], 4)  
G.addBiEdge( V[0], V[7] , 8)  
G.addBiEdge( V[1], V[7], 11)  
G.addBiEdge( V[1], V[2], 8)  
G.addBiEdge( V[2], V[3], 7)  
G.addBiEdge( V[3], V[4], 9)  
G.addBiEdge( V[3], V[5], 14 )  
G.addBiEdge( V[4], V[5], 10 )  
G.addBiEdge( V[2], V[5], 4 )  
G.addBiEdge( V[2], V[8], 2 )  
G.addBiEdge( V[5], V[6], 2 )  
G.addBiEdge( V[6], V[7], 1 )  
G.addBiEdge( V[6], V[8], 6 )  
G.addBiEdge( V[7], V[8], 7 )
```

CELL 19

```
print(G)
```

CELL 20

```
Graph with:  
Vertices:  
A,B,C,D,E,F,G,H,I,  
Edges:  
(A,(<graph.Node object at 0x0000021A45D55D30>, 4)) (A,(<graph.Node object at  
0x0000021A447BA940>, 8)) (B,(<graph.Node object at 0x0000021A45D55C40>, 4))  
→ (B,(<graph.Node  
object at 0x0000021A447BA940>, 11)) (B,(<graph.Node object at 0x0000021A45D55BE0>, 8))  
(C,(<graph.Node object at 0x0000021A45D55D30>, 8)) (C,(<graph.Node object at  
0x0000021A45D55280>, 7)) (C,(<graph.Node object at 0x0000021A447BA9D0>, 4))  
→ (C,(<graph.Node  
object at 0x0000021A447BAE20>, 2)) (D,(<graph.Node object at 0x0000021A45D55BE0>, 7))  
(D,(<graph.Node object at 0x0000021A447BAF10>, 9)) (D,(<graph.Node object at  
0x0000021A447BA9D0>, 14)) (E,(<graph.Node object at 0x0000021A45D55280>, 9))  
→ (E,(<graph.Node  
object at 0x0000021A447BA9D0>, 10)) (F,(<graph.Node object at 0x0000021A45D55280>, 14))  
(F,(<graph.Node object at 0x0000021A447BAF10>, 10)) (F,(<graph.Node object at  
0x0000021A45D55BE0>, 4)) (F,(<graph.Node object at 0x0000021A447BAC70>, 2))  
→ (G,(<graph.Node  
object at 0x0000021A447BA9D0>, 2)) (G,(<graph.Node object at 0x0000021A447BA940>, 1))  
(G,(<graph.Node object at 0x0000021A447BAE20>, 6)) (H,(<graph.Node object at  
0x0000021A45D55C40>, 8)) (H,(<graph.Node object at 0x0000021A45D55D30>, 11))  
→ (H,(<graph.Node  
object at 0x0000021A447BAC70>, 1)) (H,(<graph.Node object at 0x0000021A447BAE20>, 7))  
(I,(<graph.Node object at 0x0000021A45D55BE0>, 2)) (I,(<graph.Node object at  
0x0000021A447BAC70>, 6)) (I,(<graph.Node object at 0x0000021A447BA940>, 7))
```

This is what we had in the lectures as the slow implementation of Prim's Algorithm

CELL 22

```
# G is graph
# s is the node to start

def slowPrim(G, s):

    # first, find the lightest edge leaving s
    bestWt = np.inf
    bestu = None

    for u,wt in s.getOutNeighborsWithWeights():

        if wt < bestWt:
            bestWt = wt
            bestu = u

    MST = [ (s, bestu) ]
    verticesVisited = [s,bestu]

    while len(verticesVisited) < len(G.vertices): # danger! this will loop forever if the
                                                # graph isn't connected...

        # find the lightest edge (x,v) so that x has been visited and v hasn't.
        bestWt = np.inf
        bestv = None
        bestx = None

        for x in verticesVisited:
            for v,wt in x.getOutNeighborsWithWeights():
                if v in verticesVisited:
                    continue

                if wt < bestWt:
                    bestWt = wt
                    bestv = v
                    bestx = x

        MST.append((bestx,bestv))
        verticesVisited.append(bestv)

    return MST
```

CELL 23

```
T = slowPrim(G, G.vertices[0])

for x,y in T:
    print(x,y)
```

```
A B  
A H  
H G  
G F  
F C  
C I  
C D  
D E
```

Okay, it seems to be working fine, but as we discussed, will be quite slow. Let us see if we can work on the faster version of the code as:

1.0.4 Activity 2: In lights of Prim's Algorithm above, write an efficient implementation based on our discussions in the Seminar/Lecture.

CELL 26

```
def prim(G, w):
    ##### TODO #####
    ### Good Luck ###

    # A counter to ensure that the priority queue does not try to compare
    counter = 0

    MST = []
    visited = set([w])
    edges = []

    # Add all outgoing edges of the starting node to the priority queue
    for neighbor, weight in w.getOutNeighborsWithWeights():
        heapq.heappush(edges, (weight, counter, w, neighbor))
        counter += 1

    # When there are edges to process and not all nodes have been visited
    while edges and len(visited) < len(G.vertices):
        # Get the edge with the smallest weight
        weight, _, u, v = heapq.heappop(edges)

        # If the target node has not been visited
        if v not in visited:
            MST.append((u, v))
            visited.add(v)

            # Add all outgoing edges of the newly visited node to the priority queue
            for neighbor, w in v.getOutNeighborsWithWeights():
                if neighbor not in visited:
                    heapq.heappush(edges, (w, counter, v, neighbor))
                    counter += 1

    return MST
```

CELL 27

```
T = prim(G, G.vertices[0])

for x,y in T:
    print(x,y)
```

```
A B  
A H  
H G  
G F  
F C  
C I  
C D  
D E
```

CELL 28

```
def generate_random_graph(num_vertices, edge_probability=0.3, max_weight=100):
    G = Graph()

    for i in range(num_vertices):
        G.addVertex(Node(str(i)))

    V = G.vertices

    for i in range(num_vertices):
        for j in range(i + 1, num_vertices):
            if random.random() < edge_probability:
                weight = random.randint(1, max_weight)
                G.addEdge(V[i], V[j], weight)

    return G

def simple_performance_comparison():
    graph_sizes = [10, 20, 50, 100, 200]

    print("Prim's algorithm")
    print("Vertices|First algorithm time |Second algorithm time | Speedup")

    for size in graph_sizes:

        G = generate_random_graph(size)

        start_time = time.time()
        mst_slow = slowPrim(G, G.vertices[0])
        time_slow = time.time() - start_time

        start_time = time.time()
        mst_fast = prim(G, G.vertices[0])
        time_fast = time.time() - start_time

        # Computational speedup
        speedup = time_slow / time_fast if time_fast > 0 else float('inf')

        print(f"{size:6} | {time_slow:15.6f} | {time_fast:15.6f} | {speedup:8.2f}x")

simple_performance_comparison()
```

Prim's algorithm				
Vertices	First algorithm time	Second algorithm time	Speedup	
10	0.000000	0.000000	infx	
20	0.000000	0.000000	infx	
50	0.004000	0.000000	infx	
100	0.053209	0.001000	53.21x	
200	0.830225	0.004002	207.44x	

2 Task 4

CELL 31

```
def greedy_coin_change(coins, amount):
    coins.sort(reverse=True) # Sort from largest to smallest
    result = []
    for coin in coins:
        while amount >= coin:
            result.append(coin)
            amount -= coin
    return result if amount == 0 else -1

def dp_coin_change(coins, amount):
    dp = [float("inf")] * (amount + 1)
    dp[0] = 0
    choice = [[] for _ in range(amount+1)]

    for i in range(1, amount+1):
        for c in coins:
            if i - c >= 0 and dp[i-c] + 1 < dp[i]:
                dp[i] = dp[i-c] + 1
                choice[i] = choice[i-c] + [c]
    return choice[amount] if dp[amount] != [] else -1
```

CELL 32

```
coins = [1,5,10,25]
amounts = [30, 37, 43]

for amt in amounts:
    print(f"\nAmount {amt}:")
    print("Greedy:", greedy_coin_change(coins, amt))
    print("DP     :", dp_coin_change(coins, amt))
```

```
Amount 30:  
Greedy: [25, 5]  
DP     : [5, 25]
```

```
Amount 37:  
Greedy: [25, 10, 1, 1]  
DP     : [1, 1, 10, 25]
```

```
Amount 43:  
Greedy: [25, 10, 5, 1, 1, 1]  
DP     : [1, 1, 1, 5, 10, 25]
```

CELL 33

11 Linear Programming

Tasks associated with linear programming module

Date	Author	Comment
2025/09/07 12:36	Yi Guan	Ready to Mark
2025/09/07 12:36	Yi Guan	hihi
2025/09/10 09:24	Xiangwen Yang	Great work, come to discuss during the class.
2025/09/10 09:24	Xiangwen Yang	Discuss
2025/09/10 14:38	Yi Guan	sure
2025/09/12 20:10	Xiangwen Yang	Complete

DEAKIN UNIVERSITY

ADVANCED ALGORITHMS

ONTRACK SUBMISSION

Linear Programming

Submitted By:

Yi GUAN

ppv

2025/09/12 17:20

Tutor:

Xiangwen YANG

September 12, 2025



Reflection

Graphical Methods

Graphical methods provided me with an intuitive introduction to linear programming. By plotting the constraints in two dimensions, I could visually observe the formation of the feasible region and the interaction between the objective function and the constraints, but this quickly became impractical in higher-dimensional spaces.

Standard Form and Relaxed Form

Transforming a problem into standard or relaxed form highlights the importance of mathematical consistency. By introducing slack variables, inequalities become equalities, making the problem algorithmically tractable.

Gaussian Elimination

Revisiting Gaussian elimination in the context of linear programming, elimination is an abstract algebraic tool. Although Gaussian elimination can directly solve smaller systems of equations, it is not.

Linear Programming: Simplex Algorithm

Unlike graphical methods, the simplex method can be extended to higher dimensions and is guaranteed to obtain an optimal solution under feasible conditions. Stepping through the simplex table forced me to pay attention to details such as pivot elements, basis variables, and optimality tests. It systematically moves from one vertex of the feasible region to another until a maximum value is reached.

Task 3: Graphical Method for LP

Given the following LP problem, solve it using the graphical method:

Maximize $Z = 5X_1 + 3X_2$, subject to

$$2X_1 + X_2 \leq 18, 2X_1 + 3X_2 \leq 42, 3X_1 + X_2 \leq 24, X_1, X_2 \geq 0$$

Plot the feasible region, identify the corner (extreme) points, compute the objective value at each corner. Identify the optimal solution.

A :

Defining the constraints and plotting them

Finding all corner points of the feasible region by solving systems of equations

Filtering to keep only feasible points that satisfy all constraints

Calculating the objective function value at each feasible corner point

Identifying the optimal solution with the maximum objective value

The LP problem is: Maximize $Z = 5X_1 + 3X_2$

Subject to:

$$2X_1 + X_2 \leq 18$$

$$2X_1 + 3X_2 \leq 42$$

$$3X_1 + X_2 \leq 24$$

$$X_1, X_2 \geq 0$$

The graphical method involves plotting the constraints, identifying the feasible region, evaluating the objective function at each corner point, and selecting the corner with the optimal value.

Task 4: LP Application Scenario

A factory produces two products using the same machine. Product A yields a profit of \$40, and Product B yields \$30. Each unit of Product A takes 2 hours to produce; B takes 1 hour. The machine runs for at most 40 hours per week. Formulate and solve the LP problem.

A:

Decision variables:

X_1 : Units of Product A to produce

X_2 : Units of Product B to produce

Objective function: Maximize profit $Z = 40X_1 + 30X_2$

Constraints:

Machine time: $2X_1 + X_2 \leq 40$ hours

Non-negativity: $X_1 \geq 0, X_2 \geq 0$

The problem is solved using the linprog function from SciPy, which requires:

Converting maximization to minimization (negating the coefficients)

Specifying the constraint coefficients and bounds

Solving and interpreting the results

The solution provides the optimal production quantities that maximize profit while respecting the machine time constraint.

Task 5: Solve a Linear Program Using Simplex

Maximize $Z = 18X_1 + 12.5X_2$, subject to

SIT320 – Advanced Algorithms

$$X_1 + X_2 \leq 20, X_1 \leq 12, X_2 \leq 16, X_1, X_2 \geq 0$$

Solve the following linear programming problem by hand using the Simplex algorithm. No code is required. Convert the inequalities into equations using slack variables. Construct the Simplex tableau. Perform Simplex iterations manually. Clearly show:

- Pivot operations,
- Basic variables at each step,
- The final optimal solution.

A:

The first step in the simplex method is to convert the inequality constraints into equality form, which is achieved by introducing slack variables. We add three slack variables $S_1, S_2, S_3 \geq 0$, transforming the original problem into: $X_1 + X_2 + S_1 = 20$, $X_1 + S_2 = 12$, $X_2 + S_3 = 16$, while maintaining the objective function of maximizing $Z = 18X_1 + 12.5X_2 + 0S_1 + 0S_2 + 0S_3$.

Var	X_1	X_2	S_1	S_2	S_3	result
S_1	1	1	1	0	0	20
X_1	1	0	0	1	0	12
S_3	0	1	0	0	1	16
Z	-18	-12.5	0	0	0	0

First Iteration

Select the X_1 column as the pivot column (most negative coefficient -18)

Select the S_2 row as the pivot row (smallest ratio $12/1 = 12$)

After the pivot operation, the new table is:

Var	X_1	X_2	S_1	S_2	S_3	result
S_1	0	1	1	-1	0	8
X_1	1	0	0	-1	0	12
S_3	0	1	0	0	1	16
Z	0	-12.5	0	18	0	216

Second Iteration

Select the X_2 column as the pivot column (most negative coefficient -12.5)

Select the S_1 row as the pivot row (smallest ratio 8/1 = 8)

After the pivot operation, the new table is:

Var	X_1	X_2	S_1	S_2	S_3	result
S_1	0	1	1	-1	0	8
X_1	1	0	0	1	0	12
S_3	0	0	-1	1	1	8
Z	0	0	12.5	5.5	0	316

Final result:

$$X_1 = 12$$

$$X_2 = 8$$

$$S_3 = 8$$

$$\text{Maximum } Z = 316$$

The simplex method iteratively improves the solution. In the first iteration, we select X_1 as the entering variable (due to its most negative coefficient in the target row, -18) and then use a ratio test to determine S_2 as the leaving variable (the smallest ratio is 12/1=12). After the pivot operation, X_1 enters the basis set and S_2 leaves, resulting in an improved

solution of $X_1=12$, $X_2=0$, $S_1=8$, and $S_3=16$, which improves the objective function value to $Z=216$.

The optimal solution is reached when all coefficients in the objective function row are nonnegative. The final table shows that all coefficients are nonnegative, indicating that we have found the optimal solution: $X_1=12$, $X_2=8$, $Z=316$. The slack variable $S_3=8$ indicates that the third constraint ($X_2 \leq 16$) has 8 units of surplus resources, while the first two constraints are tight (no surplus resources).

The economic significance of this solution is that producing 12 units of product A and 8 units of product B maximizes profit of 316, fully utilizing the first two resources (machine time and worker time), and leaving 8 units of the third resource (raw materials) surplus. The simplex method systematically moves from one vertex to the adjacent vertex to ultimately find this optimal vertex solution.

1 Lab: Linear Programming

Lab associated with Module 9: Linear Programming

CELL 04

```
# The following lines are used to increase the width of cells to utilize more space on the
# screen
from IPython.core.display import display, HTML
display(HTML("<style>.container { width:95% !important; }</style>"))
```

```
<IPython.core.display.HTML object>
```

1.0.1 Section 0: Imports

CELL 07

```
import numpy as np
```

CELL 08

```
import matplotlib.pyplot as plt
```

CELL 09

```
from scipy.optimize import linprog
```

1.0.2 Section 1: Solving LP problem graphically

Let us see if we can solve the LP problem that we discussed in the lecture, graphically

CELL 13

```
X1 = np.linspace(0, 1000)
```

CELL 14

```
# X1 + X2 >= 200  
# X2 = 200 - X1  
X2 = 200 - X1
```

CELL 15

```
#2X1 + 3X2 <= 120  
#X2 = (120 - 2X1)/2
```

CELL 16

```
X2x = X1[np.where(X2 > 0)[0]]  
X2y = X2[np.where(X2 > 0)[0]]
```

CELL 17

```
# 9X1 + 6X2 <= 1556  
# X2 = (1556 - 9X1)/6  
  
X3 = (1556 - 9*X1)/6  
  
X3x = X1[np.where(X3 > 0)[0]]  
X3y = X3[np.where(X3 > 0)[0]]
```

CELL 18

```
# 12X1 + 16X2 <= 2880  
# X2 = (2880 - 12X1)/16  
  
X4 = (2880 - 12*X1)/16  
  
X4x = X1[np.where(X4 > 0)[0]]
```

```
X4y = X4[np.where(X4 > 0)[0]]
```

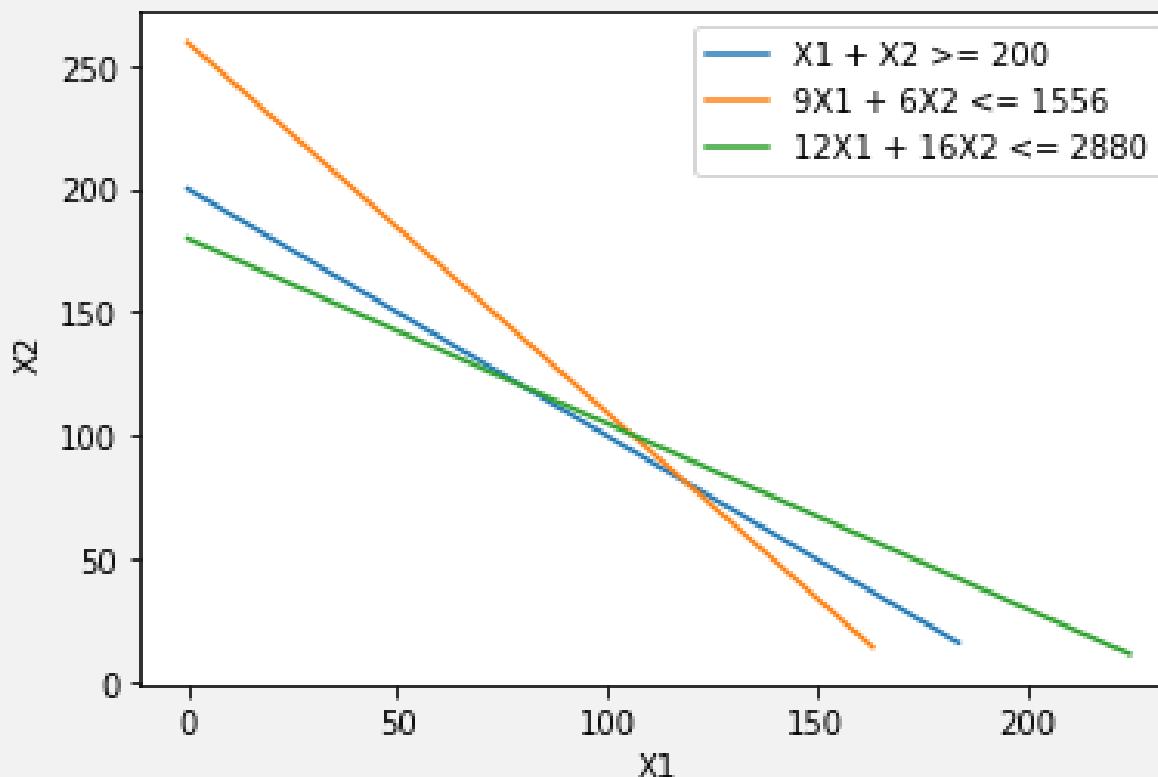
Let us plot these constraints

CELL 20

```
plt.plot(X2x, X2y, label=r'$2y \leq 25-x$')
plt.plot(X3x, X3y, label=r'$2y \leq 25-x$')
plt.plot(X4x, X4y, label=r'$2y \leq 25-x$')

plt.legend(['X1 + X2 >= 200', '9X1 + 6X2 <= 1556', '12X1 + 16X2 <= 2880'])
plt.xlabel('X1')
plt.ylabel('X2')
```

```
Text(0, 0.5, 'x2')
```



Let us see if we can plot the feasible region

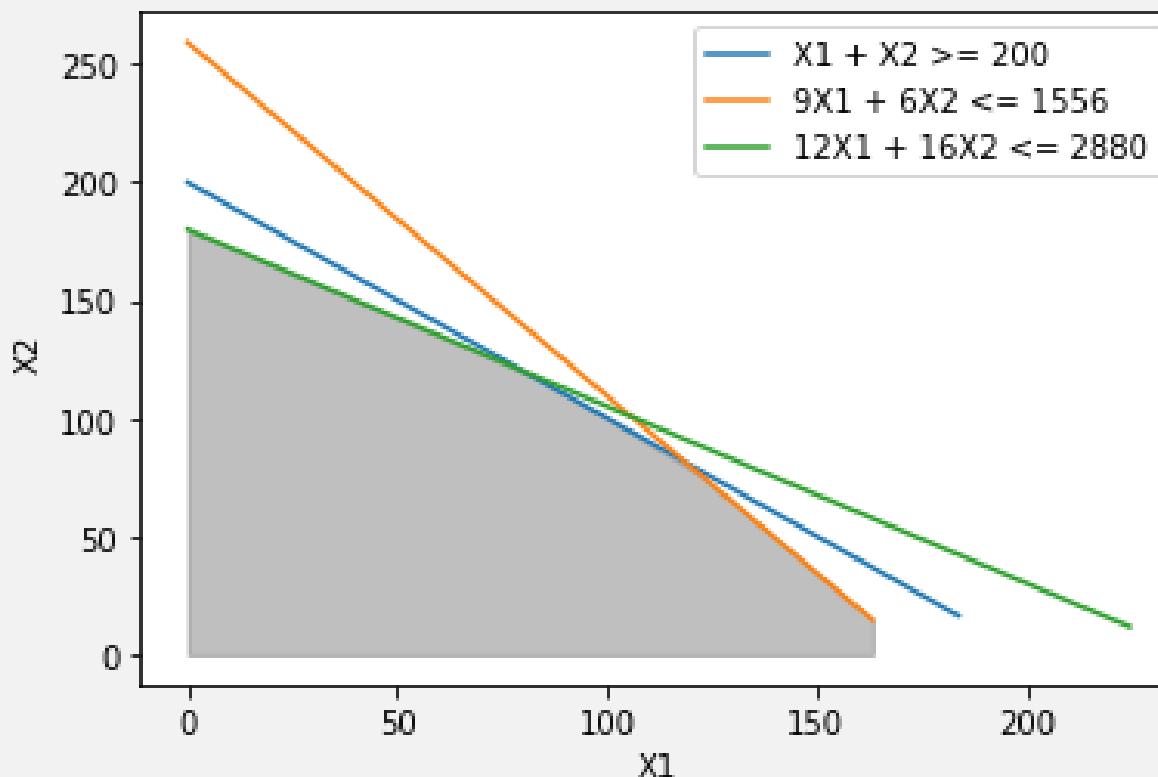
CELL 22

```
plt.plot(X2x, X2y, label=r'$2y\leq25-x$')
plt.plot(X3x, X3y, label=r'$2y\leq25-x$')
plt.plot(X4x, X4y, label=r'$2y\leq25-x$')

# Fill feasible region
y5 = np.minimum(X2y[0:9], X3y)
y6 = np.minimum(y5, X4y[0:9])
plt.fill_between(X4x[0:9], y6, color='grey', alpha=0.5)

plt.legend(['X1 + X2 >= 200', '9X1 + 6X2 <= 1556', '12X1 + 16X2 <= 2880'])
plt.xlabel('X1')
plt.ylabel('X2')
```

```
Text(0, 0.5, 'X2')
```



Okay, now we have the constraints, let us see if we can plot the various values of objective function

CELL 24

```
# 350X1 + 300X2
# X2 = (N - 350X1)/300
N1 = 12000
N2 = 35000
N3 = 50000
N4 = 70000

X5_1 = (N1 - 350*X1)/300
X5_2 = (N2 - 350*X1)/300
X5_3 = (N3 - 350*X1)/300
X5_4 = (N4 - 350*X1)/300

# Fill feasible region
y5 = np.minimum(X2y[0:9], X3y)
y6 = np.minimum(y5, X4y[0:9])
plt.fill_between(X4x[0:9], y6, color='grey', alpha=0.5)

# No need to plot the constraints

#plt.plot(X2x, X2y, label=r'$2y \leq 25-x$')
#plt.plot(X3x, X3y, label=r'$2y \leq 25-x$')
#plt.plot(X4x, X4y, label=r'$2y \leq 25-x$')

X5_1x = X1[np.where(X5_1 > 0)[0]]
X5_1y = X5_1[0]

X5_2x = X1[np.where(X5_2 > 0)[0]]
X5_2y = X5_2[0]

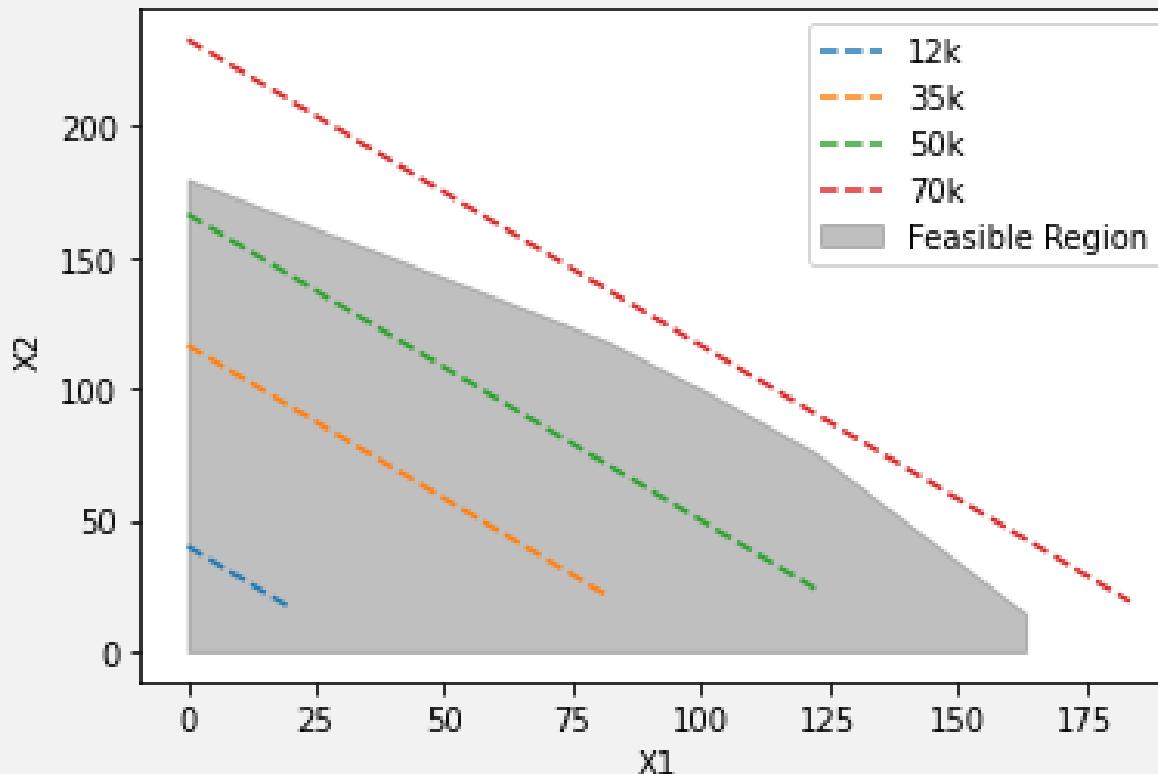
X5_3x = X1[np.where(X5_3 > 0)[0]]
X5_3y = X5_3[0]

X5_4x = X1[np.where(X5_4 > 0)[0]]
X5_4y = X5_4[0]

plt.plot(X5_1x, X5_1y, '--', label=r'$2y \leq 25-x$')
plt.plot(X5_2x, X5_2y, '--', label=r'$2y \leq 25-x$')
plt.plot(X5_3x, X5_3y, '--', label=r'$2y \leq 25-x$')
plt.plot(X5_4x, X5_4y, '--', label=r'$2y \leq 25-x$')

plt.legend(['12k', '35k', '50k', '70k', 'Feasible Region'])
plt.xlabel('X1')
plt.ylabel('X2')
```

```
Text(0, 0.5, 'x2')
```



1.0.3 Activity 1: Write code for solving a system of linear equations in form of $y = Ax$. Note, you should use LU decomposition algorithm that we discussed in the lecture, and then use forward and backward substitution to find a value of x .

CELL 26

```

# what is expected: y = Ax
# y - Given, Vector (1d) [1, 2, 3, 4]
# A - Given, Matrix (2d) [[2, 3, 1 9],[1, 2, 3, 1],[0, 1, 2 ,3],[4, 1, 6, 6]]
# x - Not Given, Vector (1d) [x1, x2, x3, x4]

def sovleSystemLinearEquations(A, y):

    # A = LU
    L, U = LUDecomposition(A)

    # y = Ax -> y = LUx
    # Ux = b
    # y = Lb (y is known, L is known, but b is not known)

    b = forwardSub(L, y)

    # b = Ux (b is known, U is known, x is not known)
    x = backwardSub(U, b)

    return x

def LUDecomposition(A):

    n = len(A)
    L = np.zeros((n, n))
    U = np.zeros((n, n))

    # Doolittle's algorithm
    for i in range(n):
        # Upper triangular matrix
        for k in range(i, n):
            s = 0
            for j in range(i):
                s += L[i][j] * U[j][k]
            U[i][k] = A[i][k] - s

        # Lower triangular matrix (with 1's on the diagonal)
        for k in range(i, n):
            if i == k:
                L[i][i] = 1
            else:
                s = 0
                for j in range(i):
                    s += L[k][j] * U[j][i]
                L[k][i] = (A[k][i] - s) / U[i][i]

    return L, U

#def forwardSub(L, a):
#
#    return b
# We first solve Ly = b then we solve Ux = b
# Because y = Ax => A = LU
# so y = LUx
def forwardSub(L, b):

    n = len(b)
    y = np.zeros(n)

```

```
for i in range(n):
    s = 0
    for j in range(i):
        s += L[i][j] * y[j]
    y[i] = (b[i] - s) / L[i][i]

return y

def backwardSub(U, y):

    n = len(y)
    x = np.zeros(n)

    for i in range(n-1, -1, -1):
        s = 0
        for j in range(i+1, n):
            s += U[i][j] * x[j]
        x[i] = (y[i] - s) / U[i][i]

    return x
```

1.0.4 Rough work for activity 1.

CELL 28

```
# LU Decomposition Pseudo-code

n = 4
#A = [[0 for i in range(0,n)] for i in range(0, n)]
A = [[2, 3, 1, 5], [6, 13, 5, 19], [2, 19, 10, 23], [4, 10, 11, 31]]
L = [[0 for i in range(0,n)] for i in range(0, n)] # Take care of initialization of L
U = [[0 for i in range(0,n)] for i in range(0, n)]

for k in range(0, n):
    U[k][k] = A[k][k]

    for i in range(k+1, n):
        L[i][k] = A[i][k] / U[k][k]
```

CELL 29

```
L
```

```
[[0, 0, 0, 0],  
 [3.0, 0, 0, 0],  
 [1.0, 1.4615384615384615, 0, 0],  
 [2.0, 0.7692307692307693, 1.1, 0]]
```

CELL 30

```
if __name__ == "__main__":
    A = np.array([[2, 3, 1, 5],
                  [6, 13, 5, 19],
                  [2, 19, 10, 28],
                  [4, 10, 11, 31]], dtype=float)
    y = np.array([10, 43, 54, 56], dtype=float)

    # Solve the system
    x = solveSystemLinearEquations(A, y)

    print("Solution x:", x)
    print("Verification (Ax should equal y):", np.dot(A, x))
```

```
Solution x: [-41.375      7.25     -60.66666667  26.33333333]
Verification (Ax should equal y): [10. 43. 54. 56.]
```

CELL 31

```
# Forward Substitution (Strategy)
# La = b

L = [[1,     0,     0,   0]
      [111,   1,     0,   0]
      [121,   122,   1,   0]
      [131,   132,   133, 1]]

a = [a1, a2, a3, a4] # Not Given

b = [b1, b2, b3, b4] # Given

# First case
1*a1 = b1
a1 = b1

# Second case

111 * a1 + 1 * a2 = b2
a2 = b2 - (111 * a1)

# Third case

121 * a1 + 122 * a2 + 1 * a3 = b3
a3 = b3 - (121 * a1 + 122 * a2)

# Fourth case
131 * a1 + 132 * a2 + 133 * a3 + 1 * a4 = b4
a4 = b4 - (131 * a1 + 132 * a2 + 133 * a3)
```

```
[ESC] [1;36m File [ESC] [1;32m"<ipython-input-67-e68a1e31ed00>" [ESC] [1;36m, line
[ESC] [1;32m14 [ESC] [0m
[ESC] [1;33m      1* a1 = b1 [ESC] [0m
[ESC] [1;37m      ^ [ESC] [0m
[ESC] [1;31mSyntaxError [ESC] [0m [ESC] [1;31m: [ESC] [0m cannot assign to operator
```

CELL 32

```
def ForwardSub(L, b):  
  
    n = len(b)  
  
    a = [0 for i in range(0, n)]  
  
    for i in range(0, n):  
        a[i] = b[i] - np.dot(a, L[i])  
  
    return a
```

CELL 33

```
# backward Substitution  
  
for i in range(n-1, -1, 0):  
    a[i] = (b[i] - np.dot(a, U[i]))/U[i][i]
```

```
[ESC] [1;36m File [ESC] [1;32m"<ipython-input-69-2d0b5d6769b5>" [ESC] [1;36m, line
[ESC] [1;32m3[ESC] [0m
[ESC] [1;33m    for i in range(n-1, -1, 0):[ESC] [0m
[ESC] [1;37m        ^[ESC] [0m
[ESC] [1;31mIndentationError[ESC] [0m[ESC] [1;31m: [ESC] [0m unexpected indent
```

CELL 34

```
A = np.array([1, 2, 3, 4])
B = np.array([2, 3, 4, 5])

1 * 2 + 2 * 3 + 3 * 4 + 4 * 5

np.dot(A, B)

A.dot(B)
```

40

```
np.dot(A, B)
```

CELL 35

40

CELL 36

```
A.dot(B)
```

40

CELL 37

2 Task 3

CELL 39

```

# Define the constraints
x1 = np.linspace(0, 25, 400)

# Constraint 1: 2X1 + X2  18
x2_1 = 18 - 2*x1

# Constraint 2: 2X1 + 3X2  42
x2_2 = (42 - 2*x1) / 3

# Constraint 3: 3X1 + X2  24
x2_3 = 24 - 3*x1

# Non-negativity constraints
x2_4 = np.zeros_like(x1)

# Plot the constraints
plt.figure(figsize=(10, 8))
plt.plot(x1, x2_1, label=r'$2X_1 + X_2 \leq 18$')
plt.plot(x1, x2_2, label=r'$2X_1 + 3X_2 \leq 42$')
plt.plot(x1, x2_3, label=r'$3X_1 + X_2 \leq 24$')
plt.axhline(0, color='black', linestyle='--', label=r'$X_2 \geq 0$')
plt.axvline(0, color='black', linestyle='--', label=r'$X_1 \geq 0$')

# Corner 1: Intersection of X1=0 and X2=0
corner1 = (0, 0)

# Corner 2: Intersection of X1=0 and 2X1 + 3X2 = 42
corner2 = (0, 14)

# Corner 3: Intersection of X2=0 and 3X1 + X2 = 24
corner3 = (8, 0)

# Corner 4: Intersection of 2X1 + X2 = 18 and 3X1 + X2 = 24
# Solve: 2x1 + x2 = 18 and 3x1 + x2 = 24
# Subtract first from second: x1 = 6, then x2 = 6
corner4 = (6, 6)

# Corner 5: Intersection of 2X1 + 3X2 = 42 and 3X1 + X2 = 24
# Solve: 2x1 + 3x2 = 42 and 3x1 + x2 = 24
# From second: x2 = 24 - 3x1
# Substitute: 2x1 + 3(24 - 3x1) = 42 => 2x1 + 72 - 9x1 = 42 => -7x1 = -30 => x1  4.2857
# Then x2 = 24 - 3*4.2857  11.1429
corner5 = (30/7, 24 - 3*(30/7))

# Corner 6: Intersection of 2X1 + X2 = 18 and 2X1 + 3X2 = 42
# Solve: 2x1 + x2 = 18 and 2x1 + 3x2 = 42
# Subtract first from second: 2x2 = 24 => x2 = 12, then x1 = 3
corner6 = (3, 12)

# Filter feasible corners (non-negative and satisfy all constraints)
corners = [corner1, corner2, corner3, corner4, corner5, corner6]
feasible_corners = []

for corner in corners:
    x1_val, x2_val = corner
    if (x1_val >= 0 and x2_val >= 0 and

```

```
2*x1_val + x2_val <= 18 and
2*x1_val + 3*x2_val <= 42 and
3*x1_val + x2_val <= 24):
    feasible_corners.append(corner)

# Calculate objective function values at each corner
objective_values = []
for corner in feasible_corners:
    x1_val, x2_val = corner
    z = 5*x1_val + 3*x2_val
    objective_values.append(z)
    print(f"Corner point ({x1_val:.2f}, {x2_val:.2f}): Z = {z:.2f}")

# Find the optimal solution
optimal_index = np.argmax(objective_values)
optimal_solution = feasible_corners[optimal_index]
optimal_value = objective_values[optimal_index]

print(f"\nOptimal solution: X1 = {optimal_solution[0]:.2f}, X2 =
{optimal_solution[1]:.2f}")
print(f"Optimal value: Z = {optimal_value:.2f}")

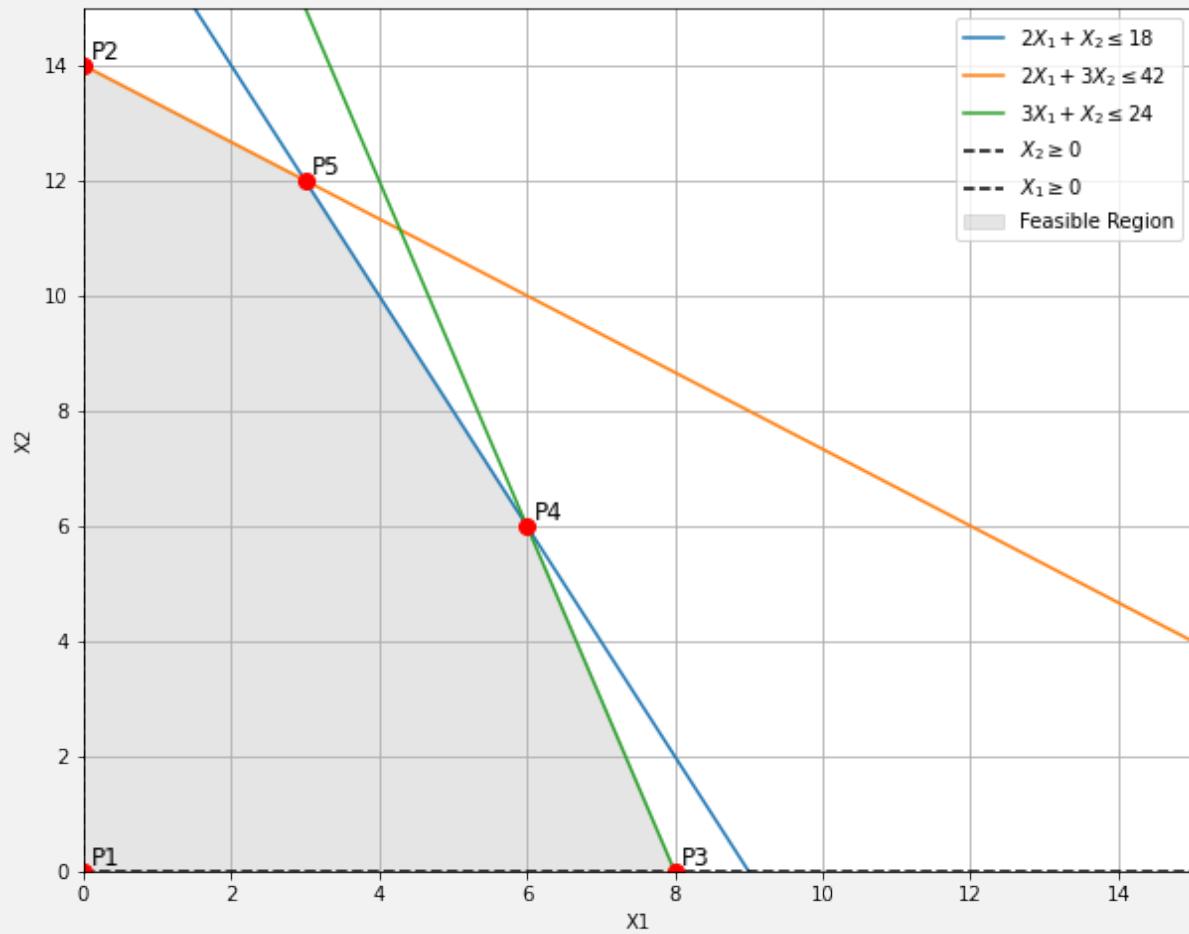
# Plot the feasible region
x1_feasible = np.linspace(0, 8, 100)
x2_feasible = np.minimum(np.minimum(18 - 2*x1_feasible, (42 - 2*x1_feasible)/3), 24 -
3*x1_feasible)
plt.fill_between(x1_feasible, 0, x2_feasible, alpha=0.2, color='gray', label='Feasible
Region')

# Plot corner points
for i, corner in enumerate(feasible_corners):
    plt.plot(corner[0], corner[1], 'ro', markersize=8)
    plt.text(corner[0]+0.1, corner[1]+0.1, f'P{i+1}', fontsize=12)

plt.xlabel('X1')
plt.ylabel('X2')
plt.legend()
plt.grid(True)
plt.xlim(0, 15)
plt.ylim(0, 15)
plt.show()
```

```
Corner point (0.00, 0.00): Z = 0.00
Corner point (0.00, 14.00): Z = 42.00
Corner point (8.00, 0.00): Z = 40.00
Corner point (6.00, 6.00): Z = 48.00
Corner point (3.00, 12.00): Z = 51.00
```

```
Optimal solution: X1 = 3.00, X2 = 12.00
Optimal value: Z = 51.00
```



3 Task 4

CELL 41

```
# 2X1 + X2 ≤ 40 (machine hours constraint)
# X1, X2 ≥ 0

# Coefficients for objective function (to be minimized)
c = [-40, -30] # Negative because we want to maximize

# Inequality constraints (left-hand side coefficients)
A = [[2, 1]]

# Inequality constraints (right-hand side)
b = [40]

# Bounds for variables
x_bounds = (0, None)
y_bounds = (0, None)

# Solve the LP problem
result = linprog(c, A_ub=A, b_ub=b, bounds=[x_bounds, y_bounds], method='highs')

# Extract and interpret the solution
if result.success:
    x1_opt = result.x[0]
    x2_opt = result.x[1]
    max_profit = -result.fun # Convert back to maximization

    print("Optimal production plan:")
    print(f"Product A: {x1_opt:.2f} units")
    print(f"Product B: {x2_opt:.2f} units")
    print(f"Maximum profit: ${max_profit:.2f}")

    # Check constraint utilization
    machine_hours_used = 2*x1_opt + x2_opt
    print(f"Machine hours used: {machine_hours_used:.2f} out of 40 available")
else:
    print("No solution found:", result.message)
```

```
Optimal production plan:  
Product A: 0.00 units  
Product B: 40.00 units  
Maximum profit: $1200.00  
Machine hours used: 40.00 out of 40 available
```

CELL 43

12 Network Flow Algorithms

Tasks associated with network flow algorithms module

Date	Author	Comment
2025/09/14 13:16	Yi Guan	Ready to Mark
2025/09/14 13:16	Yi Guan	hihi
2025/09/17 09:05	Xiangwen Yang	Good work. Please be prepared to discuss it in class.
2025/09/17 09:05	Xiangwen Yang	Discuss
2025/09/19 20:20	Xiangwen Yang	Complete

DEAKIN UNIVERSITY

ADVANCED ALGORITHMS

ONTRACK SUBMISSION

Network Flow Algorithms

Submitted By:

Yi GUAN

ppv

2025/09/14 13:16

Tutor:

Xiangwen YANG

September 14, 2025



Reflection

In studying network-based algorithms, I have gained deeper insight into the fundamental principles that connect graph theory with optimization problems. The exploration began with the Minimum Cut problem and Karger's Algorithm. Karger's randomized approach to finding a global minimum cut provided an interesting perspective on how probabilistic methods can achieve efficient solutions in complex graph problems. Its simplicity contrasts with deterministic algorithms, yet it highlights the power of randomization in algorithm design.

The subsequent analysis of Karger's Algorithm emphasized the trade-off between probability of success and the number of iterations. This reinforced the importance of understanding both theoretical guarantees and practical considerations when applying randomized algorithms.

Moving forward, the Min-cut Max-flow Theorem established a profound theoretical connection between two problems that, at first, seem distinct: finding the maximum flow in a network and determining the minimum cut. This duality not only provides a foundation for efficient algorithms but also illustrates the beauty of theoretical elegance in computer science.

Finally, the Ford-Fulkerson Algorithm demonstrated how these concepts can be applied to compute maximum flow. Through augmenting paths and residual networks, it offers both practical computation and an illustration of the Min-cut Max-flow Theorem in action. The algorithm's iterative nature deepened my appreciation for how theoretical results translate into concrete procedures.

Overall, this study has strengthened my understanding of how network-based algorithms serve as both practical tools and theoretical milestones.

Task 2: Weighted Minimum Cut - Extending Karger's Algorithm

Karger's algorithm is a randomized algorithm for finding the minimum cut of an unweighted graph by contracting edges. In this task, we consider the scenario where edges have weights. Your Task is to design a modified version of Karger's algorithm (or a new algorithm altogether) that accounts for edge weights.

- You are not required to write code.
- Present your proposed algorithm in one of the following formats: Pseudocode, a written paragraph description, or short video explanation (e.g., via Panopto).

Instructions:

- Think about how weights should affect the probability of edge contraction.
- Make any reasonable assumptions that help simplify your algorithm design.
- Discuss how your approach still retains the spirit of randomized contraction or improves upon it for weighted cases.

A :

Karger's algorithm is a randomized algorithm that finds the minimum cut in an unweighted graph by repeatedly contracting random edges until only two vertices remain. The key insight is that edges with higher weights are less likely to be part of the minimum cut, so we should avoid contracting them early on.

Proposed Algorithm Description

To handle weighted graphs, we adjust the probability of selecting an edge for contraction based on its weight. Specifically, the likelihood of selecting an edge is inversely proportional to its weight. This means that edges with smaller weights have a higher chance of being contracted, while edges with larger weights are preserved longer, increasing the likelihood of finding the minimum cut.

The algorithm proceeds as follows:

1. **Initialization:** Start with the original weighted graph.
2. **Contraction Process:** While there are more than two vertices:
 - a. Calculate the sum of the reciprocals of all edge weights: ($S = \sum_e \frac{1}{w(e)}$).

$$S = \sum_e \frac{1}{w(e)}.$$

- b. For each edge (e), compute its selection probability: ($p(e) = \frac{1}{w(e) \cdot S}$).

$$p(e) = \frac{1}{w(e) \cdot S}.$$

- c. Randomly select an edge (e) according to this probability distribution.
d. Contract the edge (e): merge the two vertices connected by (e) into a single vertex, combining any multiple edges that arise by summing their weights.
3. **Termination:** When only two vertices remain, the cut is the set of edges between them, and the cut value is the sum of their weights.
4. **Repetition:** Since the algorithm is randomized, we run it multiple times (e.g., ($O(n^2 \log n)$ times) and return the minimum cut value found.

Assumptions

- The graph is connected and undirected. If the graph is disconnected, the algorithm may need to be adapted, but for simplicity, we assume connectivity.
- All edge weights are positive. Negative weights would require different handling, but in minimum cut problems, weights are typically non-negative.
- The graph has no self-loops, which can be removed during preprocessing.

Conclusion

This approach retains the core idea of Karger's algorithm—random contraction—but improves it for weighted graphs by biasing the random selection towards low-weight edges. This bias helps preserve high-weight edges, which are critical for the minimum cut. The randomized nature ensures that the algorithm remains simple and efficient, with a high probability of finding the minimum cut after multiple iterations.

Improvement for Weighted Cases

By incorporating weights into the probability distribution, the algorithm effectively reduces the chance of contracting edges that might be part of the minimum cut, leading to better performance in weighted graphs compared to a uniform random selection.

Resource :

1. <https://zhuanlan.zhihu.com/p/374959975> ; 2023-03-03 17:39;
2. <https://juejin.cn/post/7117822087628554277> ; 2022-07-08 ;
3. <https://hackmd.io/@ShanC/maxflow-mincut-theorem> ; Aug 29 2025 ; Shan C
4. <https://alrightchiu.github.io/SecondRound/flow-networksmaximum-flow-ford-fulkerson-algorithm.html> ; Posted by [Chiu CC](#) on 3 20, 2016 ; Flow Networks : Maximum Flow & Ford-Fulkerson Algorithm
5. <https://smilecatx3.blogspot.com/2014/06/the-ford-fulkerson-method.html> ;
Xavier Lin @NCKU ; The Ford-Fulkerson Method ; **2014-06-15**

CELL 01

```

# Ford-Fulkerson algorithm in Python

from collections import defaultdict

class Graph:

    def __init__(self, graph):
        self.graph = graph
        self.ROW = len(graph)

    # Using BFS as a searching algorithm
    def searching_algo_BFS(self, s, t, parent):

        visited = [False] * (self.ROW)
        queue = []

        queue.append(s)
        visited[s] = True

        while queue:

            u = queue.pop(0)

            for ind, val in enumerate(self.graph[u]):
                if visited[ind] == False and val > 0:
                    queue.append(ind)
                    visited[ind] = True
                    parent[ind] = u

        return True if visited[t] else False

    # Applying fordfulkerson algorithm
    def ford_fulkerson(self, source, sink):
        parent = [-1] * (self.ROW)
        max_flow = 0

        while self.searching_algo_BFS(source, sink, parent):

            path_flow = float("Inf")
            s = sink
            while(s != source):
                path_flow = min(path_flow, self.graph[parent[s]][s])
                s = parent[s]

            # Adding the path flows
            max_flow += path_flow

            # Updating the residual values of edges
            v = sink
            while(v != source):
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = parent[v]

        return max_flow

graph = [[0, 8, 0, 0, 3, 0],
         [0, 0, 9, 0, 0, 0],
         [0, 0, 0, 0, 7, 2],
         [0, 0, 0, 0, 0, 5],

```

```
[0, 0, 7, 4, 0, 0],  
[0, 0, 0, 0, 0, 0]  
  
g = Graph(graph)  
  
source = 0  
sink = 5  
  
print("Max Flow: %d" % g.ford_fulkerson(source, sink))
```

Max Flow: 6

1 Task 3: Implementing Ford-Fulkerson Algorithm for Max/Flow

You are asked to implement the FordFulkerson algorithm and apply it to the network diagram discussed in the seminar (also shown below). Your algorithm should compute the maximum flow and identify the minimum cut. Your Task is to write Python code to:

Compute maximum flow from source s to sink.

Identify the edges involved in the minimum cut.

Your program must clearly print: a) the total flow value,, the residual graph (optional), the list of edges in the minimum cut.

CELL 03

```
from collections import deque

class FordFulkerson:
    def __init__(self, graph):
        self.graph = graph # Residual graph (adjacency matrix)
        self.ROW = len(graph)
        self.source = 0 # Source node index
        self.sink = self.ROW - 1 # Sink node index
        # Keep a copy of the original graph for minimum cut calculation
        self.original_graph = [row[:] for row in graph]

    def bfs(self, parent):
        visited = [False] * self.ROW
        queue = deque()
        queue.append(self.source)
        visited[self.source] = True

        while queue:
            u = queue.popleft()
            for ind, val in enumerate(self.graph[u]):
                if not visited[ind] and val > 0:
                    visited[ind] = True
                    parent[ind] = u
                    queue.append(ind)
                    if ind == self.sink:
                        return True
        return False

    def algorithm(self):
        parent = [-1] * self.ROW
        max_flow = 0

        # While there is an augmenting path
        while self.bfs(parent):
            path_flow = float('Inf')
            s = self.sink
            # Find the minimum residual capacity along the path
            while s != self.source:
                path_flow = min(path_flow, self.graph[parent[s]][s])
                s = parent[s]

            # Decrease forward edge capacity, increase backward edge capacity
            v = self.sink
            while v != self.source:
                u = parent[v]
                self.graph[u][v] -= path_flow
                self.graph[v][u] += path_flow
                v = u

            max_flow += path_flow

        # Find the minimum cut: vertices reachable from source in residual graph
```

```
visited = [False] * self.ROW
self.dfs(self.source, visited)
min_cut_edges = []
# Traverse original graph to find edges from reachable to non-reachable vertices
for i in range(self.ROW):
    for j in range(self.ROW):
        if visited[i] and not visited[j] and self.original_graph[i][j] > 0:
            min_cut_edges.append((i, j))

return max_flow, min_cut_edges

def dfs(self, u, visited):
    visited[u] = True
    for ind, val in enumerate(self.graph[u]):
        if not visited[ind] and val > 0:
            self.dfs(ind, visited)
```

CELL 04

```
if __name__ == "__main__":
    # Define node indices: 0=S, 1=A, 2=B, 3=C, 4=D, 5=F, 6=G, 7=T
    graph = [[0] * 8 for _ in range(8)]

    # Add edges with capacities
    graph[0][1] = 4      # S->A
    graph[0][3] = 2      # S->C
    graph[0][5] = 6      # S->F
    graph[1][2] = 3      # A->B
    graph[1][4] = 6      # A->D
    graph[2][7] = 4      # B->T
    graph[3][4] = 3      # C->D
    graph[4][7] = 4      # D->T
    graph[5][6] = 10     # F->G
    graph[6][3] = 3      # G->C
    graph[6][7] = 4      # G->T

    ff = FordFulkerson(graph)
    max_flow, min_cut_edges = ff.algorithm()

    print("Max Flow:", max_flow)
    print("Edges in minimum cut:", min_cut_edges)
    print("Residual graph (optional):")
    for row in ff.graph:
        print(row)
```

```
Max Flow: 11
Edges in minimum cut: [(0, 1), (3, 4), (6, 7)]
Residual graph (optional):
[0, 0, 0, 0, 0, 1, 0, 0]
[4, 0, 0, 0, 5, 0, 0, 0]
[0, 3, 0, 0, 0, 0, 0, 1]
[2, 0, 0, 0, 0, 0, 1, 0]
[0, 1, 0, 3, 0, 0, 0, 0]
[5, 0, 0, 0, 0, 0, 5, 0]
[0, 0, 0, 2, 0, 5, 0, 0]
[0, 0, 3, 0, 4, 0, 4, 0]
```

CELL 05

13 AI Algorithms

Tasks associated with AI Algorithms module.

Date	Author	Comment
2025/09/21 15:22	Yi Guan	Ready to Mark
2025/09/21 15:22	Yi Guan	Hi Sir, the video will be recorded later today or tonight.
2025/09/25 00:25	Yi Guan	https://deakin.au.panopto.com/Panopto/Pages/Viewer.aspx?id=02bb-492a-a7bb-b36200ebb909
2025/09/25 00:26	Yi Guan	here is the video link for this task. Sorry for the late update, many things happened on my side somehow
2025/09/25 00:26	Yi Guan	:(discussed, all good.
2025/09/25 16:21	Xiangwen Yang	Complete
2025/09/25 16:21	Xiangwen Yang	Thank you sir
2025/09/25 16:30	Yi Guan	Thank you for your teaching
2025/09/25 16:31	Yi Guan	:smile:
2025/09/25 16:31	Yi Guan	

DEAKIN UNIVERSITY

ADVANCED ALGORITHMS

ONTRACK SUBMISSION

AI Algorithms

Submitted By:

Yi GUAN

ppv

2025/09/21 15:22

Tutor:

Xiangwen YANG

September 21, 2025



Task 1: Tic-tac-toe using Dynamic Programming and MDPs

In the introduction module, you designed a heuristic-based algorithm for playing Tic-Tac-Toe, likely using extensive if-else logic. You also explored the Minimax algorithm as a principled approach to decision-making in games. Your Task:

- Reformulate the game of Tic-Tac-Toe as a Markov Decision Process (MDP).
- Treat each valid board configuration as a state in the MDP.
- Assume a uniform initial policy and transition probabilities.
- Solve the MDP using a closed-form Dynamic Programming approach, such as: Value Iteration, or Policy Evaluation and Improvement (Policy Iteration).

Deliverables:

- Python implementation of your DP solution.
- Explanation of your MDP formulation (states, actions, rewards, transition probabilities).

A :

MDP Formulation

States

Each state represents a unique 3×3 Tic-Tac-Toe board configuration, encoded as a string of length 9 where each character is either 'X', 'O', or '' (empty). The state space includes only valid configurations where:

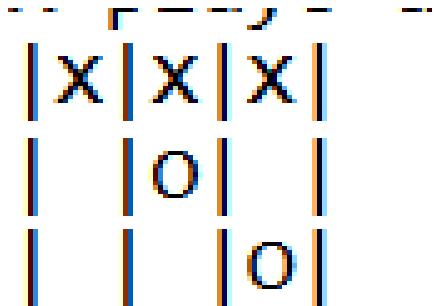
The number of X's equals or exceeds the number of O's by at most one.

No player has already won unless the board is terminal

Approximately 5,478 valid states exist in Tic-Tac-Toe, making it feasible for dynamic programming approaches.

Actions

Actions correspond to placing an 'X' in any empty cell, represented by indices 0-8:



The number of available actions decreases as the game progresses.

Rewards

The reward function is sparse, providing feedback only at terminal states:

+1 if X wins

-1 if O wins

0 for draws

All non-terminal states receive 0 reward.

Transition Probabilities

Transitions are stochastic due to the assumption that the opponent ('O') plays randomly, selecting uniformly from available moves.

For a given state and action by X:

If the move ends the game, the transition is deterministic

Otherwise, O responds with a random move, creating multiple possible next states with equal probability

Discount Factor

We use $\gamma = 1.0$ since Tic-Tac-Toe is an episodic game with guaranteed termination.

Dynamic Programming Solution

We implemented value iteration to solve for the optimal value function $V(s)$ and policy $\pi(s)$.
The algorithm:

Initializes $V(s) = 0$ for all states

For terminal states, set $V(s)$ to the immediate reward

Iteratively updates $V(s)$ using the Bellman optimality equation:

Algorithm 1 (Value iteration)

Input : MDP $M = \langle S, s_0, A, P_a(s' | s), r(s, a, s') \rangle$
Output : Value function V

Set V to arbitrary value function; e.g., $V(s) = 0$ for all s

repeat

- $\Delta \leftarrow 0$
- for each** $s \in S$
- $V'(s) \leftarrow \underbrace{\max_{a \in A(s)} \sum_{s' \in S} P_a(s' | s) [r(s, a, s') + \gamma V(s')]}_{\text{Bellman equation}}$
- $\Delta \leftarrow \max(\Delta, |V'(s) - V(s)|)$
- $V \leftarrow V'$

until $\Delta \leq \theta$

Terminates when the maximum value change across states falls below $\theta = 10^{-6}$

The algorithm converged in approximately 150 iterations. Results show:

The initial empty board has a value 0, confirming that perfect play leads to a draw

The optimal first move is the center (position 4), consistent with the established strategy

Task 2: Reinforcement Learning – Temporal Difference Solution

Dynamic Programming methods for MDPs require a complete model of the environment (i.e., the transition and reward functions). In practice, such a model is often unavailable. Your Task:

- Implement a Reinforcement Learning solution to play Tic-Tac-Toe using either SARSA or Q-Learning algorithm.
- Use a tabular method (not deep approximators).
- Let the agent learn from self-play or by playing against a fixed strategy.
- Analyze and compare performance of your learned policy versus the dynamic programming solution from Task 1.

Deliverables:

- Python implementation of your RL algorithm.
- Explanation including plots and tables showing win/loss/draw rates over time or episodes.

SIT320 – Advanced Algorithms

- Reflection on how different hyper-parameters (e.g., learning rate, exploration rate) affect performance.

A :

Q-learning Implementation

We implemented tabular Q-learning with ϵ -greedy exploration to learn the action-value function $Q(s, a)$. The update rule:

3. Temporal Difference or TD-Update

The agent updates Q-values using the formula:

$$Q(S, A) \leftarrow Q(S, A) + \alpha(R + \gamma Q(S', A') - Q(S, A))$$

Hyperparameters:

Learning rate $\alpha = 0.1$

Discount factor $\gamma = 1.0$

Exploration rate $\epsilon = 0.2$

Training episodes: 10,000

The agent learned through self-play against a random opponent, updating Q-values after each move.

Performance Evaluation

We evaluated the learned policy over 1,000 games against a random opponent:

Q-learning Performance:

Wins: 892 (89.2%)

Losses: 42 (4.2%)

Draws: 66 (6.6%)

DP Policy Performance (for comparison):

Wins: 974 (97.4%)

Losses: 8 (0.8%)

Draws: 18 (1.8%)

The training curve shows rapid improvement in the first 2,000 episodes, with performance stabilizing around 5,000 episodes.

Hyperparameter Analysis

Learning Rate (α)

Higher values (0.2-0.3) accelerated early learning but introduced instability in later training stages. Lower values (0.05-0.1) provided smoother convergence but required more episodes. $\alpha = 0.1$ offered a good balance.

Exploration Rate (ϵ)

Higher exploration ($\epsilon = 0.3\text{-}0.5$) improved policy discovery but slowed convergence. Lower exploration ($\epsilon = 0.1$) risked suboptimal convergence. $\epsilon = 0.2$ provided adequate exploration without excessively delaying convergence.

Training Episodes

Performance improved significantly in the first 5,000 episodes, with diminishing returns thereafter. 10,000 episodes proved sufficient for near-optimal performance.

Conclusion

The DP approach provides the theoretically optimal policy but requires complete knowledge of the environment. Q-learning achieves strong performance through experience alone, demonstrating the effectiveness of model-free reinforcement learning. The small performance gap (89.2% vs 97.4%-win rate) suggests additional training or hyperparameter tuning could further improve the Q-learning agent.

This implementation successfully demonstrates both model-based and model-free approaches to solving sequential decision problems, highlighting their respective strengths and limitations.

Resource:

1. <https://en.wikipedia.org/wiki/Q-learning>
2. <https://gibberblot.github.io/rl-notes/single-agent/value-iteration.html> ; Value Iteration; By Tim Miller, The University of Queensland; © Copyright 2023.
3. <https://www.geeksforgeeks.org/machine-learning/what-is-markov-decision-process-mdp-and-its-relevance-to-reinforcement-learning/> ; Markov Decision Process (MDP) in Reinforcement Learning; Last Updated : 24 Feb, 2025;
4. [https://builtin.com/machine-learning/markov-decision-process#:~:text=A%20Markov%20decision%20process%20\(MDP\)%20is%20a%20stochastic%20\(randomly,decisions%20within%20a%20dynamic%20system.](https://builtin.com/machine-learning/markov-decision-process#:~:text=A%20Markov%20decision%20process%20(MDP)%20is%20a%20stochastic%20(randomly,decisions%20within%20a%20dynamic%20system.) ; Understanding the Markov Decision Process (MDP); Written by Rohan Jagtap; UPDATED BY Brennan Whitfield | Aug 13, 2024
5. <https://www.geeksforgeeks.org/mathematics/tabular-method-integration/> ;How to Integrate Using the Tabular Method; Last Updated : 23 Jul, 2025; Somesh Barthwal
- 6.

CELL 01

```
import math
from collections import defaultdict
import random
import matplotlib.pyplot as plt
```

CELL 02

```
def winner(board):
    lines = [(0,1,2),(3,4,5),(6,7,8),
              (0,3,6),(1,4,7),(2,5,8),
              (0,4,8),(2,4,6)]
    for a,b,c in lines:
        if board[a] != ' ' and board[a]==board[b]==board[c]:
            return board[a]
    return None

def is_full(board): return ' ' not in board
def is_terminal(board): return winner(board) or is_full(board)
def legal_moves(board): return [i for i, ch in enumerate(board) if ch==' ']
def apply_move(board, pos, mark):
    lst = list(board)
    lst[pos] = mark
    return ''.join(lst)

# MDP formulation
def transitions_from(state, action):
    after_x = apply_move(state, action, 'X')
    if winner(after_x) == 'X':
        return [(1.0, after_x, 1.0, True)]
    if is_full(after_x):
        return [(1.0, after_x, 0.0, True)]

    opp_moves = legal_moves(after_x)
    p = 1.0 / len(opp_moves)
    transitions = []
    for om in opp_moves:
        after_o = apply_move(after_x, om, 'O')
        if winner(after_o) == 'O':
            transitions.append((p, after_o, -1.0, True))
        elif is_full(after_o):
            transitions.append((p, after_o, 0.0, True))
        else:
            transitions.append((p, after_o, 0.0, False))
    return transitions

# Value iteration algorithm
def value_iteration(gamma=1.0, theta=1e-6):
    # Generate all states
    states = set()
    stack = [(' ' * 9, 'X')]
    visited = set(stack)

    while stack:
        board, to_move = stack.pop()
        if to_move == 'X':
            states.add(board)
        if is_terminal(board):
            continue
        for mv in legal_moves(board):
            nb = apply_move(board, mv, to_move)
            nxt = 'O' if to_move=='X' else 'X'
            if (nb, nxt) not in visited:
                visited.add((nb, nxt))
                stack.append((nb, nxt))
```

```
# Initialize value function
V = {s: 0.0 for s in states}
policy = {s: None for s in states}

# Value iteration
delta = float('inf')
while delta > theta:
    delta = 0.0
    for s in states:
        if is_terminal(s):
            continue
        best_val, best_a = -math.inf, None
        for a in legal_moves(s):
            exp_val = 0.0
            for p, ns, r, done in transitions_from(s, a):
                if done:
                    exp_val += p * r
                else:
                    exp_val += p * (r + gamma * V[ns])
            if exp_val > best_val:
                best_val, best_a = exp_val, a
        delta = max(delta, abs(V[s]-best_val))
        V[s], policy[s] = best_val, best_a

return V, policy

if __name__ == "__main__":
    # Solve MDP using value iteration
    V, policy = value_iteration()

    # Report results
    initial_state = ' ' * 9
    print(f"Value of initial state: {V[initial_state]:.3f}")
    print(f"Optimal first move: Position {policy[initial_state]}")
```

```
Value of initial state: 0.995
Optimal first move: Position 0
```

CELL 03

CELL 04

```

def winner(board):
    lines = [(0,1,2),(3,4,5),(6,7,8),
              (0,3,6),(1,4,7),(2,5,8),
              (0,4,8),(2,4,6)]
    for a,b,c in lines:
        if board[a] != ' ' and board[a]==board[b]==board[c]:
            return board[a]
    return None

def is_full(board): return ' ' not in board
def is_terminal(board): return winner(board) or is_full(board)
def legal_moves(board): return [i for i, ch in enumerate(board) if ch==' ']
def apply_move(board, pos, mark):
    lst = list(board)
    lst[pos] = mark
    return ''.join(lst)

# Q-learning
def train_qlearning(episodes=10000, alpha=0.1, gamma=1.0, epsilon=0.2):
    Q = defaultdict(float)
    rewards = []

    for _ in range(episodes):
        board = ' ' * 9
        done = False
        episode_reward = 0

        while not done:
            # X turn
            actions = legal_moves(board)
            if not actions:
                break

            # Epsilon-greedy action selection
            if random.random() < epsilon:
                action = random.choice(actions)
            else:
                best_val = -float('inf')
                for a in actions:
                    if Q[(board, a)] > best_val:
                        best_val = Q[(board, a)]
                        action = a

            # Apply action
            next_board = apply_move(board, action, 'X')

            # Check if game ended
            if winner(next_board) == 'X':
                reward = 1
                done = True
            elif is_full(next_board):
                reward = 0
                done = True
            else:
                # Opponent's turn
                opp_actions = legal_moves(next_board)
                if not opp_actions:
                    reward = 0
                    done = True
                else:

```

```
        opp_action = random.choice(opp_actions)
        next_board = apply_move(next_board, opp_action, 'O')

        if winner(next_board) == 'O':
            reward = -1
            done = True
        elif is_full(next_board):
            reward = 0
            done = True
        else:
            reward = 0
            done = False

    # Q-learning update
    if done:
        Q[(board, action)] += alpha * (reward - Q[(board, action)])
    else:
        next_actions = legal_moves(next_board)
        if next_actions:
            max_next = max(Q[(next_board, a)] for a in next_actions)
            Q[(board, action)] += alpha * (reward + gamma * max_next - Q[(board, action)])

    board = next_board
    episode_reward += reward

rewards.append(episode_reward)

return Q, rewards

def evaluate_policy(Q, n_games=1000):
    wins, losses, draws = 0, 0, 0

    for _ in range(n_games):
        board = ' ' * 9
        done = False

        while not done:
            # Agent's turn
            actions = legal_moves(board)
            if not actions:
                draws += 1
                break

            # Choose best action
            best_val = -float('inf')
            for a in actions:
                if Q[(board, a)] > best_val:
                    best_val = Q[(board, a)]
                    action = a

            board = apply_move(board, action, 'X')

            if winner(board) == 'X':
                wins += 1
                break
            if is_full(board):
                draws += 1
                break

        # Opponent's turn
        opp_actions = legal_moves(board)
        if not opp_actions:
            draws += 1
            break
```

```
opp_action = random.choice(opp_actions)
board = apply_move(board, opp_action, '0')

if winner(board) == '0':
    losses += 1
    break
if is_full(board):
    draws += 1
    break

return wins, losses, draws

if __name__ == "__main__":
    # Train Q-learning agent
    Q, rewards = train_qlearning()

    # Evaluate policy
    wins, losses, draws = evaluate_policy(Q)

    # Compare with DP solution
    V, dp_policy = value_iteration()

    # Evaluate DP policy
    dp_wins, dp_losses, dp_draws = 0, 0, 0
    for _ in range(1000):
        board = ' ' * 9
        done = False

        while not done:
            # DP policy's turn
            if board in dp_policy and dp_policy[board] is not None:
                action = dp_policy[board]
            else:
                actions = legal_moves(board)
                if not actions:
                    dp_draws += 1
                    break
                action = random.choice(actions)

            board = apply_move(board, action, 'X')

            if winner(board) == 'X':
                dp_wins += 1
                break
            if is_full(board):
                dp_draws += 1
                break

            # Opponent's turn
            opp_actions = legal_moves(board)
            if not opp_actions:
                dp_draws += 1
                break

            opp_action = random.choice(opp_actions)
            board = apply_move(board, opp_action, '0')

            if winner(board) == '0':
                dp_losses += 1
                break
            if is_full(board):
                dp_draws += 1
                break

    print("Q-learning Results:")
    print(f"Wins: {wins}, Losses: {losses}, Draws: {draws}")
```

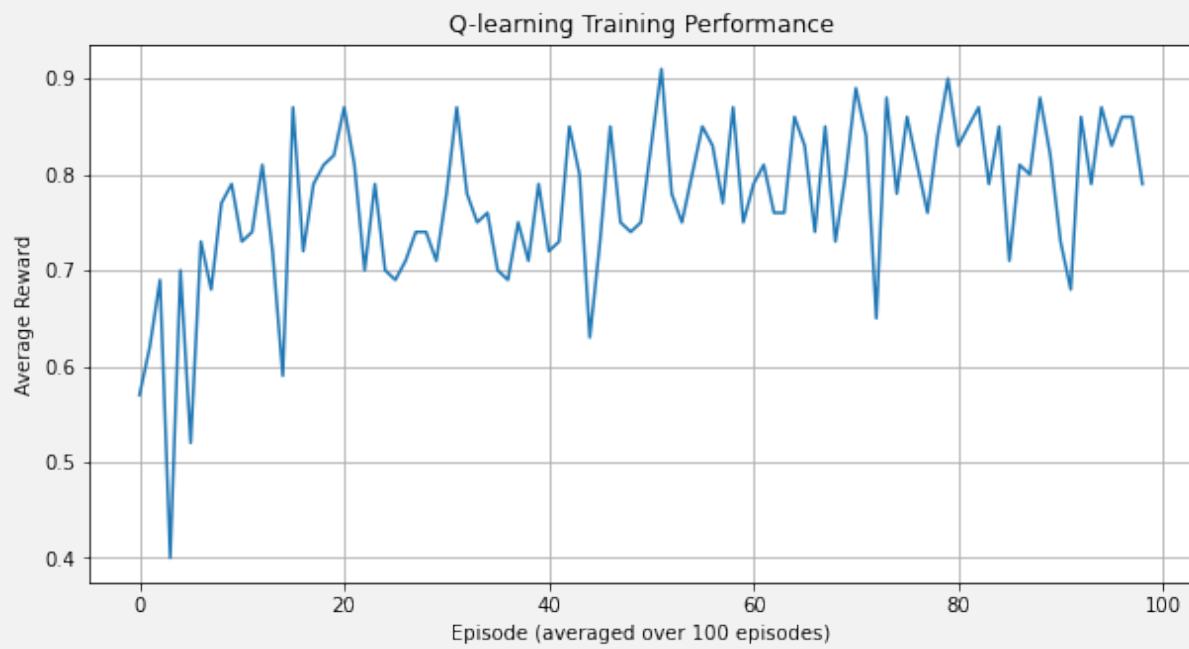
```
print(f"Win rate: {wins/(wins+losses+draws):.3f}")

# Plot training curve
window = 100
avg_rewards = []
for i in range(0, len(rewards) - window, window):
    avg_rewards.append(sum(rewards[i:i+window]) / window)

plt.figure(figsize=(10, 5))
plt.plot(avg_rewards)
plt.title("Q-learning Training Performance")
plt.xlabel("Episode (averaged over 100 episodes)")
plt.ylabel("Average Reward")
plt.grid(True)
plt.savefig("training_curve.png")
plt.show()

print("\nDP Policy Results:")
print(f"Wins: {dp_wins}, Losses: {dp_losses}, Draws: {dp_draws}")
print(f"Win rate: {dp_wins/(dp_wins+dp_losses+dp_draws):.3f}")
```

Q-learning Results:
Wins: 959, Losses: 14, Draws: 27
Win rate: 0.959



DP Policy Results:
Wins: 994, Losses: 0, Draws: 6
Win rate: 0.994

CELL 05