# Iterations

## Sliding window

Closely pay attention to some features for sliding window approach showed in Figure 1.

* Sliding window approach is working for detecting something in a continuous collection such as array, list type. Data access should support random access or support iteration.
* Window moves forward from the beginning to the end. There is not backward movements.
* The size of a window ranges from 1 or entire length of that collection. The size could be fixed or adjustable.
* A window is identified by left and right points. Size and movement is changed by changing one of or both of the two points. Usually, the left point is less than or equal to the right point.

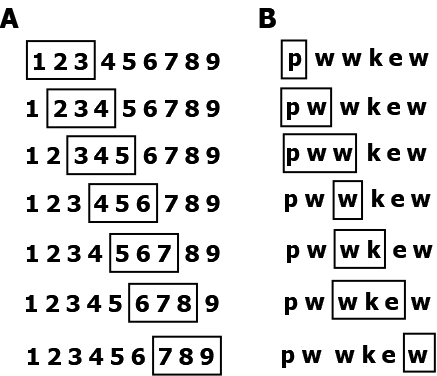


Figure . Slide windows through a collection. A. fixed-size window. B. flexible-size window.

here is an example of fixed-size window. Given an array of integers, find maximum sum subarray with a certain size.

|  |
| --- |
| [90,2,40,6,10,3,23,12]  -> [90, 2] |
| def max\_sum\_subarray(self, arr, size) -> List:  if 0 <= len(arr) <= size: return arr    max\_sum, max\_arr = 0, []  for L in range(len(arr)-size+1):  R = L + size  window\_sum = sum(arr[L:R])  if window\_sum > max\_sum:  max\_sum = window\_sum  max\_arr = arr[L:R]  return max\_arr |

Given a string s, find the length of the longest substring without repeating characters. For example, if the string is "pwwkew", the longest substring is "wke". Figure 1B. shows how sliding windows detect all substrings without repeating characters. Once sliding windows walk through all possibility, the longest one would be gotten.

|  |
| --- |
|  |
| def lengthOfLongestSubstring(self, s: str) -> int:  """  detect the length of the longest substring  """  if len(s) == 0: return 0  #string should consist of at least one character  L, R = 0, 1  max\_len = sub\_len = R - L  while L < R and R < len(s):  sub = s[L:R]  res = sub.find(s[R])  if res >= 0:  L = L+ res + 1  R += 1  sub\_len = R - L  if sub\_len > max\_len:  max\_len = sub\_len  return max\_len |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

# Recursion

## Introduction

Recursive methods is a method that call itself within method body.

**Advantages**

* Recursive functions make the code look clean and elegant.
* A complex task can be broken down into simpler sub-problems using recursion.
* recursion may make sequence generation easier than certain nested iteration.

**weakness**

* Logic behind recursion may be hard to follow through. Read recursion codes may be harder than read iteration codes.
* Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
* Complex recursive functions are hard to debug.

**features**

* recursion Direction: start from most complicated input to simplest input
* easy to get result with the simplest input
* easy to go into the next input given a current input.
* Not difficult to get relationships between the results of a current input and its next input.

How computer execute a recursive method. Here is one example: calculate multiplication of n. In math expression:

Res = n! = n\*(n-1)\*(n-2)\*…\*1 (n>=1)

There is a relationship of results between n and n-1

func(n) = n\*func(n-1)

Flow control: suppose there are n inputs. always get the next input and output based on current input and output

Call: func(n) -> func(n-1) -> ... -> func(1)

Complete: func(1)==1-> func(2)==func(1)\*2==1\*2==2-> …-> func(n)

Here is basic approach to write a recursive function.

|  |
| --- |
|  |
| def func(...):  #base case: result with simplest input  #Check to see whether the current value(s) being processed match the base case.  if ...:  return ...  #recursion  #Redefine the answer in terms of a smaller or simpler sub-problem or sub-problems.  else:  #Run the algorithm on the sub-problem.  return func(...) |

## Simple recursion

features of simple recursion:

* Walk through only one path. There is no branch. Backward walk or skip or exist in advance is not needed. For example walk through a numeric sequence 1,2,3..100 and access each element from 1 to 100 one by one.
* Any two neighboring steps could be abstracted as certain math expression. For example, accumulative multiplication could be expressed as in math expression f(n) = n\*f(n-1), n>=1
* Pass values as arguments and return values. Any changes or updates would be within one recursive step.
* In-place modification. Only update one element at a time.
* Only return final results. Don't care about intermediate results or return details of incursion.

steps:

1. play around examples using simple input

2. think about range of input, condition of exitance of recursion

3. investigate relationship between two neighbor input and their result. and then generate pattern using certain formula

### recursion of numeric sequence

accumulative sum: 1+2+...+100

sum(1)=1, sum(2)=2+ sum(1), …, **sum(n)= n + sum(n-1)**

|  |
| --- |
|  |
| def acc\_sum(n):  if n==1:  return 1  else:  return n+acc\_sum(n-1) |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

### recursion of list

Recursion could be used iterating array or list. Through walking through all elements, recursion could give some operations as the below:

* add/delete/change an element
* search/get an element
* sort list

Recursive steps of collections:

* check status for exiting: mylist==[]
* return new list using list slice: mylist[0]+mylist[1:]: (0+(1+(2+(...(n))...)

|  |
| --- |
|  |
| #one difference: for-loop locate an element by its index,  #but recursion locate the element by its neibouring element (relatively)  #iteration pattern  def print\_list\_iteration(mylist):  for i in mylist:  print(i) |
| #basic recursive pattern  #iterate a list  def print\_list(mylist):  if mylist==[]:  return []  else:  print(mylist[0])  print\_list(mylist[1:])  print\_list([1,5,8]) |

Count total number of letters if a list consists of multiple strings.

|  |
| --- |
|  |
| def count\_list(self, mylist):  if len(mylist)==0:  return 0  else:  el = mylist[0]  return len(el) + self.count\_list(mylist[1:]) |

|  |
| --- |
|  |
|  |

### recursion of a string

common issues:

* reverse a string
* find a certain character of substring from left or right side.
* count occurrence of a certain character given a string.

|  |
| --- |
|  |
| def reverse\_str(self, data):  if not isinstance(data, str):  return None  if len(data)<=1:  return data  return data[-1]+self.reverse\_str(data[:-1]) |

## Path recursion

### combination of multiple recursion

In simple recursion problems, the step switching from one stage to the next stage is constant and determined before recursion. For example, scroll a list from start to the end. The step is one, all elements would be accessed.

|  |
| --- |
| recursive(n), n = 1,2,3,4,5,6,… |
| def recursive(n):  print(n)  if n==1:  return n  else:  return recursive(n-1) |

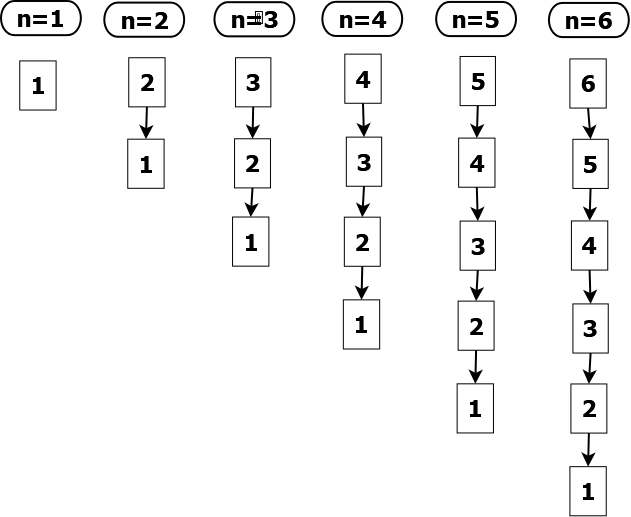


Figure . recursion of a sequence.

|  |
| --- |
| recursive(n), n=1,2,3,4,5,6,…. |
| def recursive(n):  print(n)  if n == 1:  return n  elif n == 2:  return recursive(n-1)  else:  return recursive(n-1) + recursive(n-2) |

Figure 2 shows the order of nodes after the above recursion function. That recursive process walk through all node start from root node. Walk direction is always top-down and left-right. Only access each node once. That is depth-firs search.

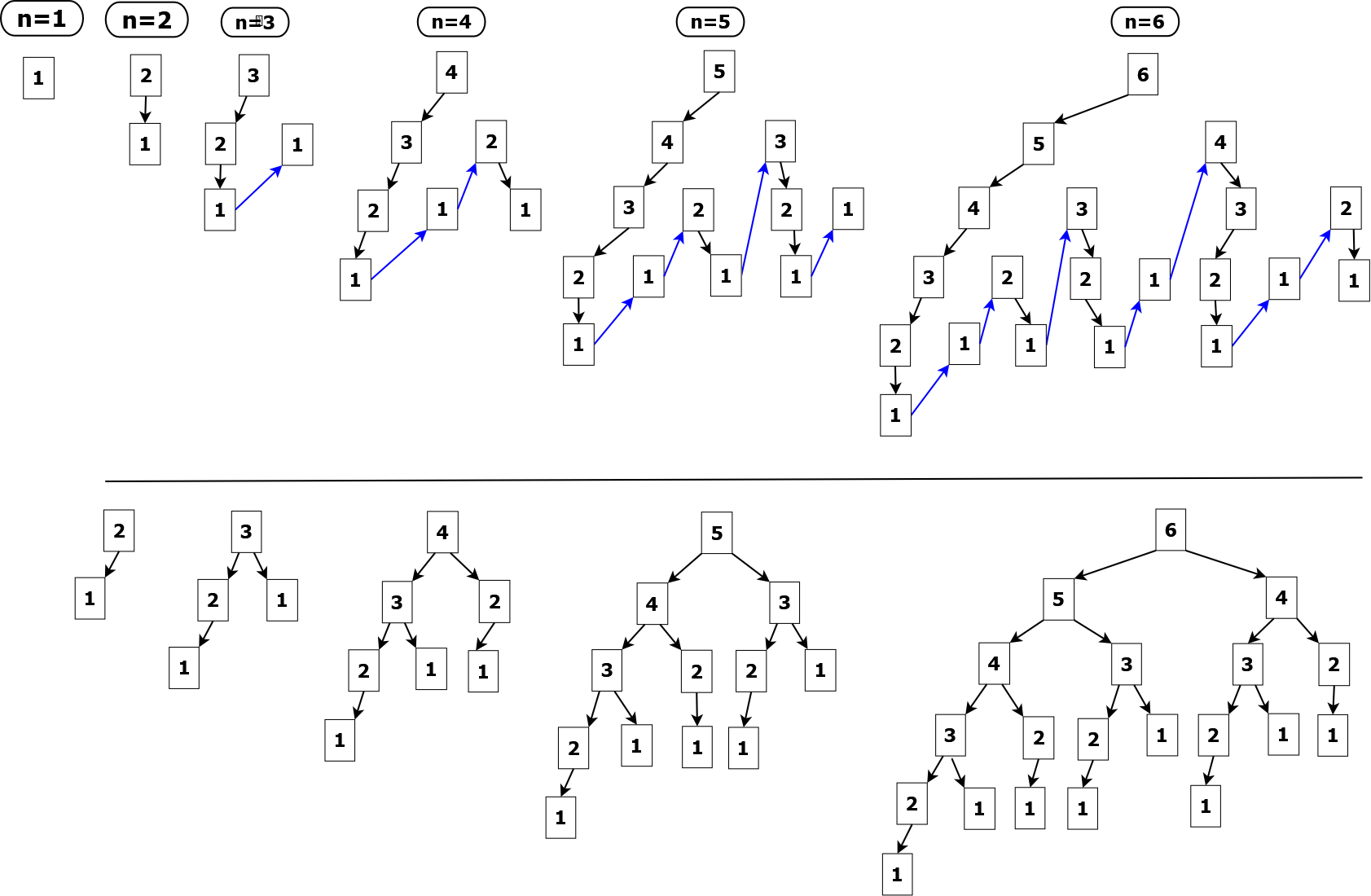


Figure . Walk through all nodes through incursion.

This recursive function actually walk through data as complete binary tree showed as Figure 3. As a complete binary tree, except end nodes, all other nodes including the root node must have a left element or a pair of elements known as left/right element.

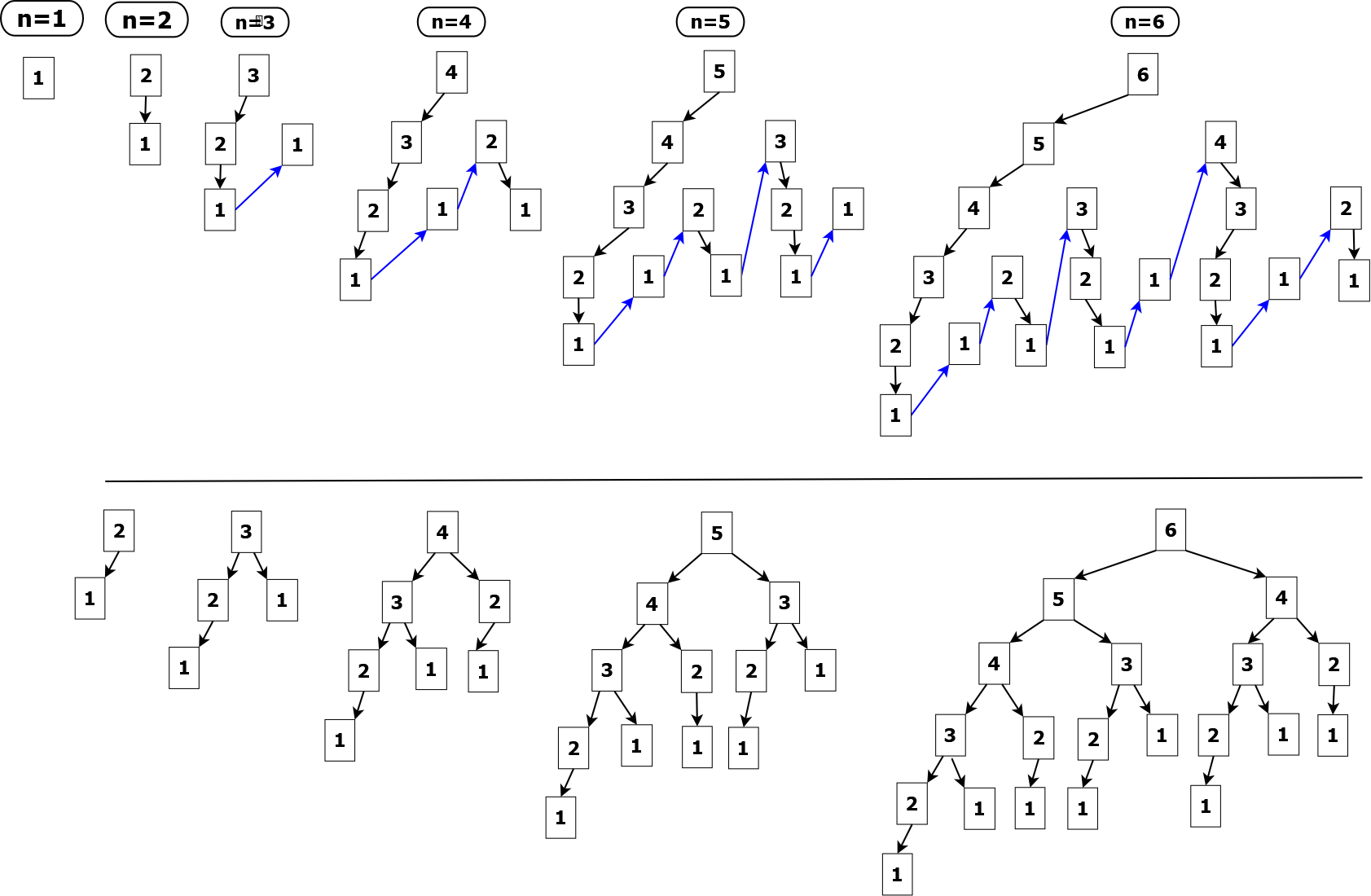


Figure . Data Structure of complete binary tree.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| root=n | Paths  up->down | Number of Ends | Number of Nodes | Number of Edges |  |
| 1 | 1 | 1 | 0+0+1=1 | 1-1=0 |  |
| 2 | 1 | 1 | 1+0+1=2 | 2-1=1 |  |
| 3 | 1+1=2 | 1+1=2 | 2+1+1=4 | 4-1=3 |  |
| 4 | 2+1=3 | 2+1=3 | 4+2+1=7 | 7-1=6 |  |
| 5 | 3+2=5 | 3+2=5 | 7+4+1=12 | 12-1=11 |  |
| 6 | 5+3=8 | 5+3=8 | 12+7+1=20 | 20-1=19 |  |
| 7 | 8+5=13 | 8+5=13 | 20+12+1=33 | 33-1=32 |  |
| 8 | 13+8=21 | 13+8=21 | 33+20+1=54 | 54-1=53 |  |
| 9 | 21+13=34 | 21+13=34 | 54+33+1=88 | 88-1=87 |  |

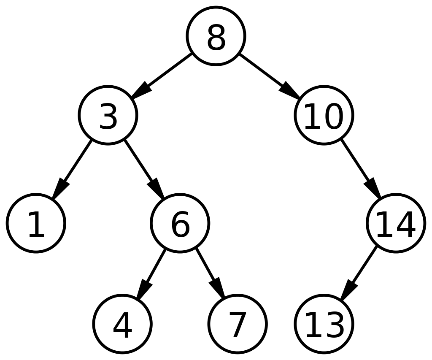
Let's say, if root is equal to n, and Nodes(1) = 1, Nodes(2) = 2, Ends(1) = Ends(2) = 1, Paths(1) = Paths(2) = 1. Therefore,

Nodes(n) = Nodes(n-1) + Nodes(n-2) + 1

Edges(n) = Nodes(n) - 1

Ends(n) = Paths(n) = Paths(n-1) + Paths(n-2) = Ends(n) + Ends(n-1)

For path recursion compared with simple recursion, the recursive path has branches. Given a node in incursion, it is possible that not a node, or single one node or two other nodes are connected with this node.



root node: 8

parent nodes: 8, 3, 10, 6, 14

sibling nodes: 3-10, 1-6, 4-7

end nodes: 1, 4, 7, 13

Simple recursion only a certain total counts once existing recursion, and no intermediate results are returned. In terms of those problems, total possible number of paths, or a shortest path, or all possible paths may be required. To meeting such expectation, paths could be stored in heap type, which pops up one element from top and adds one element from the end.

|  |
| --- |
| output:  path= [6, 5, 3, 1]  path= [6, 4, 3, 1]  path= [6, 4, 2, 1]  path= [6, 5, 4, 3, 1]  path= [6, 5, 4, 2, 1]  path= [6, 5, 3, 2, 1]  path= [6, 4, 3, 2, 1]  path= [6, 5, 4, 3, 2, 1]  total paths= 8 |
| def incur\_3(pool):  p = pool.pop(0)  n = p[-1]  if n == 1:  print('path=', p)  return n  elif n == 2:  pool.append(p + [n-1])  return incur\_3(pool)  else:  pool.append(p + [n-1])  pool.append(p + [n-2])  return incur\_3(pool) + incur\_3(pool)  #  n = 6  pool = [[n]] #pool is heap  paths = incur\_3(pool)  print(f"total paths= {paths}") |

### Step case problem

In steps problem, the steps could multiple options. For example, given N steps, a person is asked to step from the bottom onto the top. He could step out 1 or 2 or M steps at a time. The question is how many possible paths can he take to reach the top

Let's start a simple combination M=2. Therefore, the person could step out 1 or 2 steps at a time. How many possible paths can he take in order to reach the top step?

|  |
| --- |
| output:  path= [0, 1, 3, 5]  path= [0, 2, 3, 5]  path= [0, 2, 4, 5]  path= [0, 1, 2, 3, 5]  path= [0, 1, 2, 4, 5]  path= [0, 1, 3, 4, 5]  path= [0, 2, 3, 4, 5]  path= [0, 1, 2, 3, 4, 5]  Number of paths = 8 |
| def step1(pool, top):  p = pool.pop(0)  n = p[-1]  if n == top:  print('path=', p)  return n  elif n == top - 1:  pool.append(p + [n+1])  return step1(pool, top)  else:  pool.append(p + [n+1])  pool.append(p + [n+2])  return step1(pool, top) + step1(pool, top)  start, top = 0, 5  pool = [[start]] #pool is heap  steps = step1(pool, top)  paths = int(steps/top)  print(f"Number of paths = {paths}") |

The next, if the person could step out 1 or 2 or 3 steps at a time. How many possible paths can he take to reach the top step?

|  |
| --- |
| output:  path= [0, 2, 5]  path= [0, 3, 5]  path= [0, 1, 2, 5]  path= [0, 1, 3, 5]  path= [0, 1, 4, 5]  path= [0, 2, 3, 5]  path= [0, 2, 4, 5]  path= [0, 3, 4, 5]  path= [0, 1, 2, 3, 5]  path= [0, 1, 2, 4, 5]  path= [0, 1, 3, 4, 5]  path= [0, 2, 3, 4, 5]  path= [0, 1, 2, 3, 4, 5]  Number of paths = 13 |
| def step1(pool, top):  p = pool.pop(0)  n = p[-1]  if n == top:  print('path=', p)  return n  elif n == top - 1:  pool.append(p + [n+1])  return step1(pool, top)  elif n == top - 2:  pool.append(p + [n+1])  pool.append(p + [n+2])  return step1(pool, top) + step1(pool, top)  else:  pool.append(p + [n+1])  pool.append(p + [n+2])  pool.append(p + [n+3])  return step1(pool, top) + step1(pool, top) + step1(pool, top)  start, top = 0, 5  pool = [[start]] #pool is heap  steps = step1(pool, top)  paths = int(steps/top)  print(f"Number of paths = {paths}") |

|  |
| --- |
|  |
|  |

## permutation

Permutation is a string or collection with the same elements in a different order. If a string has n letters. How many combinations?

n! = n × (n-1) × (n-2) …× 2 × 1

The anagrams of a string "abc" are "abc", "acb", "bbc", "bcb", "cab", or "cba". The total number of permuted strings:

3! = 3 × 2 × 1= 6

There are several algorithm on permutation. Here, we focus on recursive permutation

Apply recursion to solve permutation issue. Consider features of permutation compared with simple recursion:

* Start point could be multiple. For example, the first letter could be 'a', 'b' or 'c'. Therefore, an iteration should be called before entering recursion.
* The sequence or collection should support random access. Walk through all possibility and identify variants by index. So string type or array type would facilitate recursion.

Firstly, how do we walk through string by iteration and recursion..

|  |
| --- |
| output: a b c d |
| def walk\_seq(s):  R = len(s)  for i in range(R):  print(i, s[i])  s="abcd"  walk\_seq(s) |
| def walk\_seq(s):  if s:  print(s[0])  return walk\_seq(s[1:])    s="abcd"  walk\_seq(s) |

|  |
| --- |
| output:  abcd  bcd  cd  d |
| def walk\_seq(s):  R = len(s)  for L in range(R):  print(s[L:])  s="abcd"  walk\_seq(s) |
| def walk\_seq(s):  if s:  print(s)  walk\_seq(s[1:])  s="abcd"  walk\_seq(s) |

permutation of string "abc"

|  |
| --- |
| output:  ['a', 'b', 'c']  ['a', 'c', 'b']  ['b', 'a', 'c']  ['b', 'c', 'a']  ['c', 'b', 'a']  ['c', 'a', 'b'] |
| def recur\_seq(s, L, R):  if L==R:  print(f"{s}")  pass  else:  for i in range(L,R+1):  s[L], s[i] = s[i], s[L]  recur\_seq(s, L+1, R)  s[L], s[i] = s[i], s[L]  def walk\_seq(s):  L, R = 0, len(s)  recur\_seq(list(s), L, R-1)  s="abc"  walk\_seq(s) |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## partition

For partitioning, we define left pointer and right pointer. The right pointer refers to pivot value. It is assumed that all values on the left of the pivot value are smaller, and all values on the right are greater. Partitioning method is very useful for sorting collections.

Figure 4. shows mov the left pointer from zero of the list one by one Compare the values of left -right pointers, if the left one is greater then switch the value. The left pointer stop moving and move back to the zero when the left pointer and right pointer are overlapped, and at the meantime the right pointer would move left. The recursion would stop when the right pointer moves to the zero.

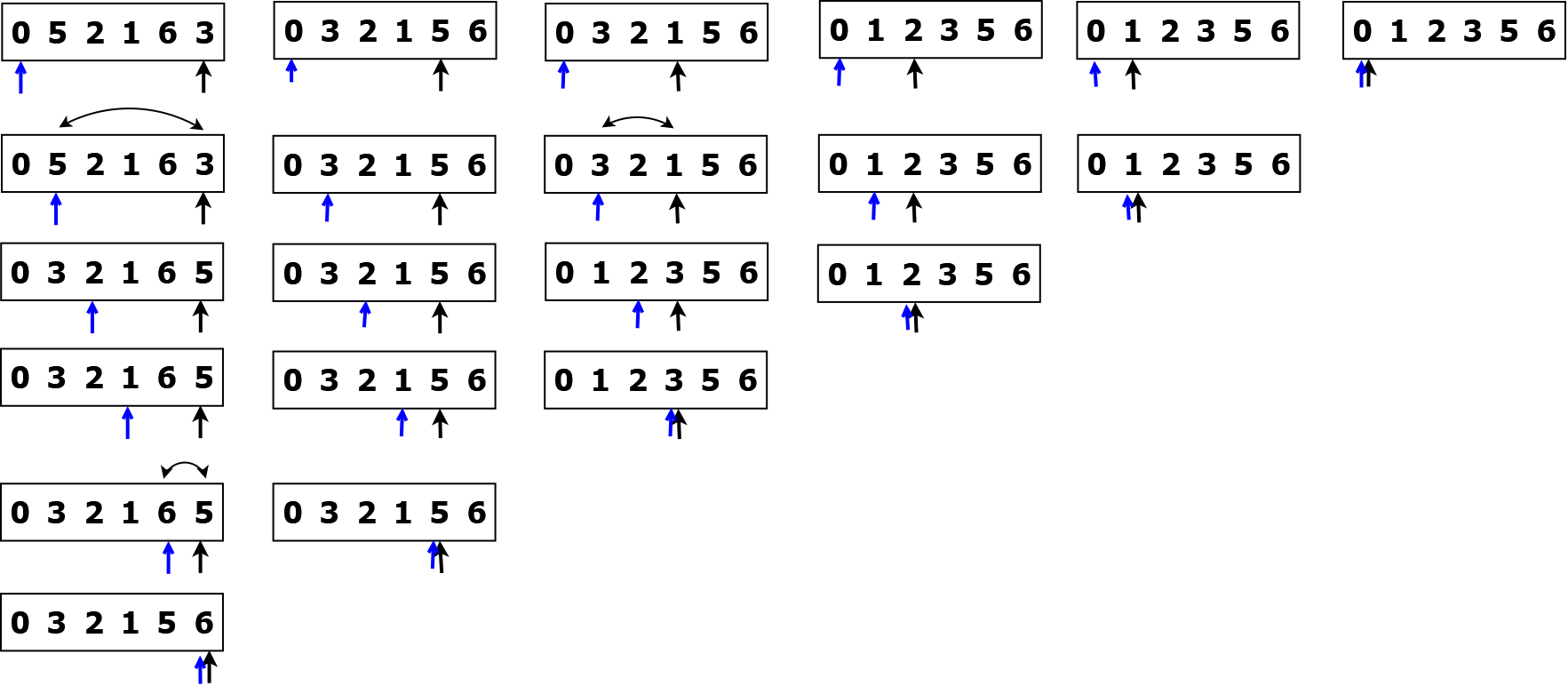


Figure . Sort list using partitioning.

Here is one recursion function, which is core function for move one element to the right place. For example, the right place is at index=5 of the list. The recursive function would move the largest value (==6) to index=5.

|  |
| --- |
| output:  [0, 5, 2, 1, 6, 3]  [0, 3, 2, 1, 5, 6] |
| def recur\_list(c, L, R):  if L < R:  if c[L]> c[R]:  c[L], c[R] = c[R], c[L]  recur\_list(c, L+1, R)  s=[0,5,2,1,6,3]  print(s)  recur\_list(s, 0, 5)  print(s) |

Here is complete source code which would sort list in ascending order.

|  |
| --- |
| output:  [0, 1, 2, 3, 5, 6, 6] |
| def recur\_list(c, L, R):  if L < R:  if c[L]> c[R]:  c[L], c[R] = c[R], c[L]  recur\_list(c, L+1, R)    def sort\_ascending(c):  if len(c) == 0:  return []  for P in range(len(c)-1, -1, -1):  recur\_list(c, 0, P)  s=[0,6,5,2,1,6,3]  sort\_list(s)  print(s) |

|  |
| --- |
|  |
|  |

## combination

A combination is how many combinations of k bins from n items.

nCr = 1, r = 0

nCr = n, r = n

nCr = (n-1)C(r-1) + (n-1)Cr

Here is the recursive code which walk through all bins with two combinations. Remember the code because that could be act as core code

|  |
| --- |
| output:  Number of end nodes = 5 |
| def recur(L, R):  if L == R:  return 1  elif L == (R-1) or L != 0:  **return recur(L+1, R)**  else:  **return recur(L+1, R) + recur(L, R-1)**    ends = recur(0,5)  print(f"Number of end nodes = {ends}") |

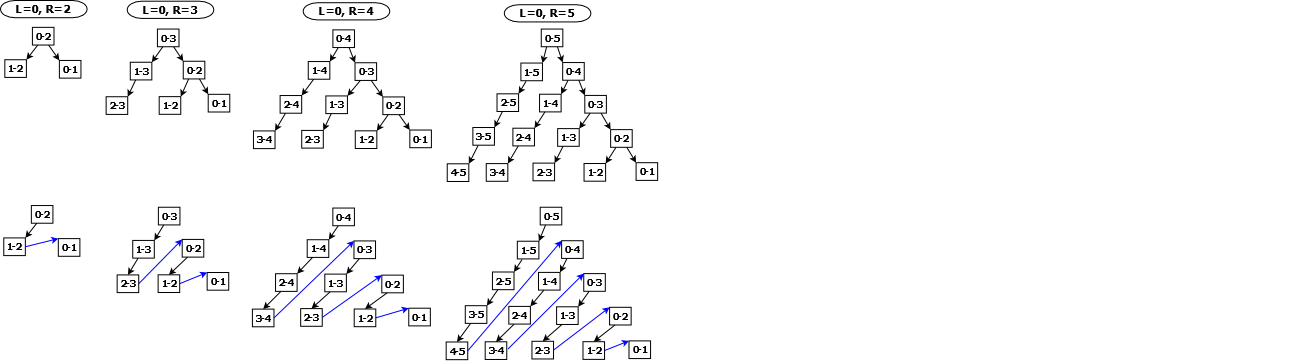


Figure . Search all combinations of 2 items from n items.

Searching all combinations of 2 from n (n>=2) items is the same as walking through a complete binary tree. In this tree, except the most right node, the other nodes have only left leaves.

The recursive function below shows how can we retrieve all combinations if n=5, k=2.

|  |
| --- |
| output:  combinations=10: [(1, 5), (2, 5), (3, 5), (4, 5), (1, 4), (2, 4), (3, 4), (1, 3), (2, 3), (1, 2)]  Number of end nodes = 4 |
| def recur(s, L, R, pool):  if L == R:  return 1  elif L == (R-1) or L != 0:  pool.append((s[L], s[R]))  return recur(s, L+1, R, pool)  else:  pool.append((s[L], s[R]))  return recur(s, L+1, R, pool) + recur(s, L, R-1, pool)  def combinations(s):  if len(s)==0:  return [[s]]  L, R = 0, len(s)-1  pool = []  ends = recur(s, L,R, pool)  print(f"combinations={len(pool)}: {pool}")  print(f"Number of end nodes = {ends}")  #  items = [1,2,3,4,5]  combinations(items) |

How can we count all combinations. for example, if n=6, k could be 0-6.

|  |
| --- |
| output:  n=6, k=0, combinations=1  n=6, k=1, combinations=6  n=6, k=2, combinations=15  n=6, k=3, combinations=20  n=6, k=4, combinations=15  n=6, k=5, combinations=6  all combinations of n=6 is 63. |
| def recur(n, r):  if r == 0 or n==r:  return 1  else:  return recur(n-1, r) + recur(n-1, r-1)  def count\_combinations(n):  res = 0  for i in range(n):  c = recur(n, i)  print(f"n={n}, k={i}, combinations={c}")  res += c  print(f"all combinations of n={n} is {res}.")  count\_combinations(6) |

The recursive function below export a Pascal's Triangle.

|  |
| --- |
| output:  1  1 1  1 2 1  1 3 3 1  1 4 6 4 1  1 5 10 10 5 1 |
| def recur(n, r):  if r==0 or n==r:  return 1  else:  return recur(n-1, r) + recur(n-1, r-1)  def pascal\_triangle(rows):  for r in range(rows):  for c in range(r+1):  comb = recur(r, c)  print(f"{comb}", end="\t")  print("")  #  pascal\_triangle(6) |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## Two points: contrary moving

Given a collection, namely array, list or string, two pointers refer to the first and end of position. Walk through the entire collection by moving left pointer forward and right pointer backward until the pointers are meeting.

|  |
| --- |
| output:  0-6  1-5  2-4  3  1-12  2-11  3-10  4-9  5-8  6-7 |
| def recur(L, R):  if L == R:  print(f"{L}")  elif L == (R - 1):  print(f"{L}-{R}")  else:  print(f"{L}-{R}")  recur(L+1, R-1)  recur(0,6)  recur(1,12) |

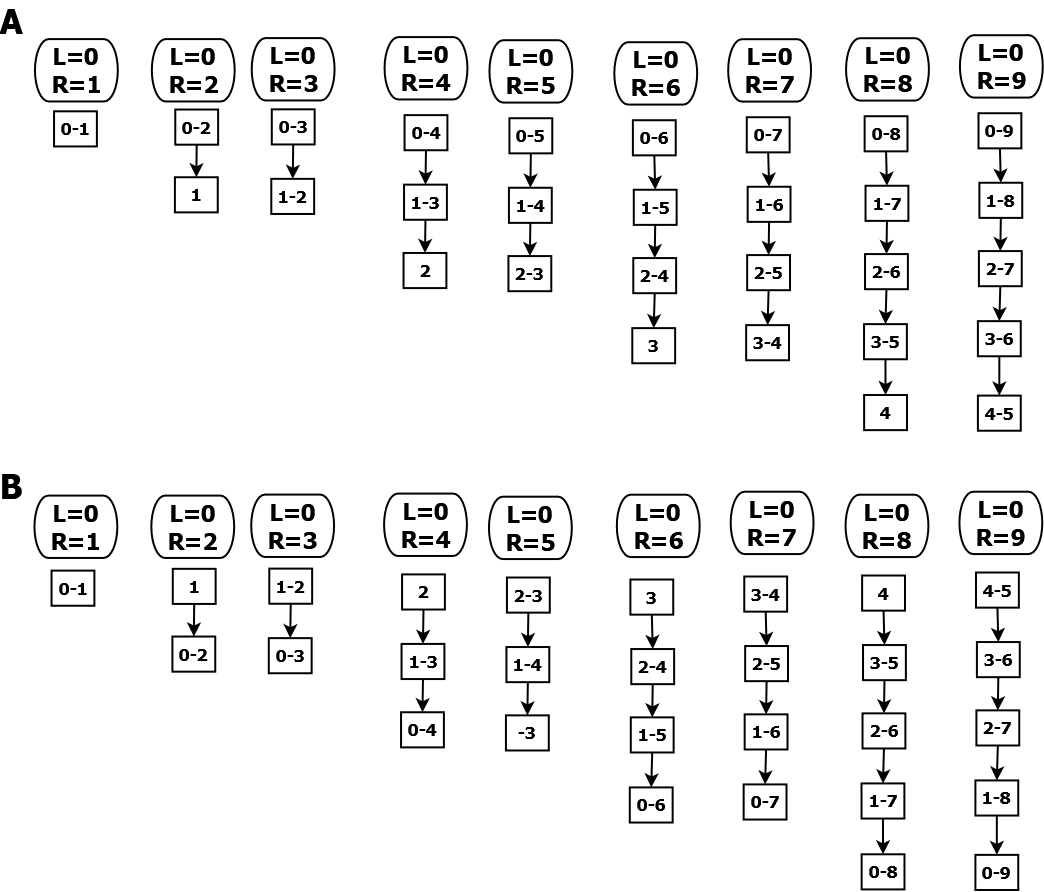


Figure . Moving two pointers by face-to-face. A. move by face-to-face. B. move by back-to-back.

Figure 6. shows how move two pointers. This approach has some features:

* Two pointers are defined. They usually point to the two ends or the middle position of a certain collection.
* There is only one moving path, but the directions of them are contrary.
* the two pointers shall move together with same pace(usually one step one node).

This approach is widely used for solving some problems, such as reverse a array, or detect palindrome string.

The example function below shows how to reverse a list in-place. This example shows some features for list operation:

* In-place modification by switching values. That saves memory usage.
* Two iterators: Locate two elements by two indexes at a time. The approach of two iterators is widely used in permutation.
* Movement of the two indexes follows a rule. Here, the two indexes start from the two end and move face to face.

|  |
| --- |
|  |
| def switch(self, mylist, a, b):  if a<b and 0 <= a < len(mylist) and 0 <= b < len(mylist):  mylist[a], mylist[b] = mylist[b], mylist[a]  if a<b:  self. switch(mylist, a+1, b-1)    def reverse\_list(self, mylist):  if len(mylist) > 1:  start = 0  end = len(mylist) -1  self.switch(mylist, start, end)  return mylist |

Here is example for detecting palindrome string namely "racecar", "abccba". The first method is based on the rule that the reverse string of palindromic string is equal to itself. The second method is based on recursive approach of two pointers.

|  |
| --- |
|  |
| def is\_palindrome1(s):  if len(s)> 1 :  return bool(s[::-1] == s)  return False |
| def is\_palindrome2(s):  if len(s) == 1:  return True  elif len(s) >= 2:  if s[0] == s[-1]:  if len(s) == 2:  return True  return is\_palindrome2(s[1:-1])  return False |

In practice, searching palindromic string have some features:

* Two possibility: odd length for example "aba", or even length namely "abba".
* Usually the string should be longer than 3.
* Palindromic string is mirror like. There is a seed string. For example, the seed string of "abccba" is "cc", and the seed string of "abcdcba" is "d" or "cdc".
* One approach is to expand seed string at two side at a time. For example, "cc" -> "bccb" ->"abccba". Therefore, only check the beginning and ending character in each recursion.
* Usually consider the longest palindromic sub-string give a long string.
* Exclude repeat string such as "aaaaa"

The code below shows how to detect the long palindromic substrings give a string.

|  |
| --- |
|  |
| from typing import List  class RecursionString:  def \_\_init\_\_(self):  self.target = []  def detect\_palindrome(self, s:str, seed: int):  """  #extend palindrome sub-string as long as it can  #input\_str[start:end] must be palindrome  """  if seed > 2 and len(s) >= seed:  #determine the best seed  for L in range(len(s)-2):  if s[L] == s[L+1]:  self.expand\_palindrome(s, L, L+1)  if s[L] == s[L+2]:  self.expand\_palindrome(s, L, L+2)  return self.target  def expand\_palindrome(self, s, L, R):  if L > 0 and R < len(s)-1:  ns = s[(L-1):(R+2)]  if ns == ns[::-1]:  return self.expand\_palindrome(s, L-1, R+1)  if len(self.target) > 0:  current = self.target[0]  if R-L > current[1] - current[0]:  self.target = [(L, R, s[L:R+1])]  if R-L == current[1] - current[0]:  self.target.append((L, R, s[L:R+1]))  else:  self.target = [(L, R, s[L:R+1])] |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## Two points: binary search

Binary search is based on movement of two points. The code below shows the rule of movements of the two points in binary recursion.

|  |
| --- |
|  |
| def recur\_left(L, R):  if L <= R:  M = int((L+R)/2)  print(f"{L}-{M}-{R}")  recur\_left(L, M-1)  def recur\_right(L, R):  if L <= R:  M = int((L+R)/2)  print(f"{L}-{M}-{R}")  recur\_right(M+1, R) |

Figure 7. shows how binary search walk through a collection. Compared with iteration all elements from the beginning to the end, binary search would consume less iteration. The two points L and R could determine the middle point. No matter which region is selected, the middle point should be compared with the target value in each recursion.

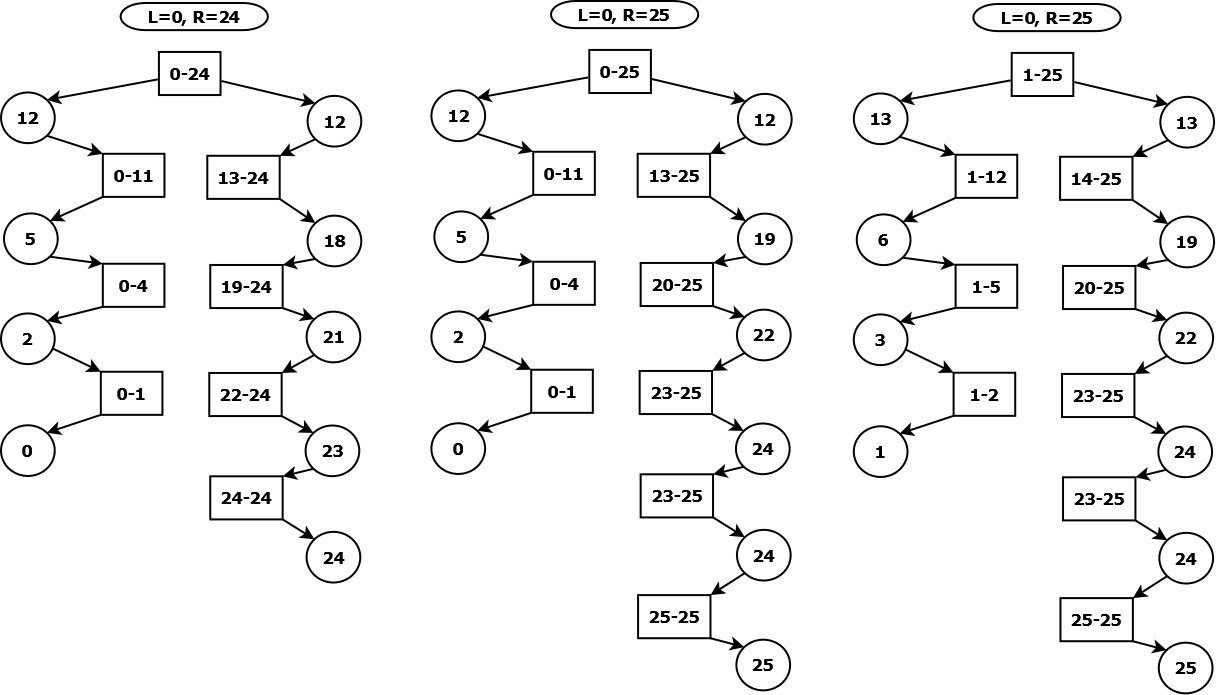


Figure . Binary search: retrieve middle points in recursive approach. The values in rectangle denote left and right boundary. The values in circles represent middle points.

binary search a value in a sorted list. Return the most right index if one value is matched or return -1.

* The list should be sorted in ascending.
* The list may consist of duplicate values.

|  |
| --- |
|  |
| class Solution:  def search\_value(self, s: List, target):  """  binary search sorted list: return index or -1  """  if len(s) == 0:  return -1  L, R = 0, len(s)-1  return self.binary\_search(s, L, R, target)  def binary\_search(self, s, L, R, target):  if L > R:  return -1  M = int((L+R)/2)  if s[M] == target:  if M == 0 or (M > 0 and s[M-1] != target):  return M  elif s[M] < target:  return self.binary\_search(s, M+1, R, target)  return self.binary\_search(s, L, M-1, target) |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## chaining

That is single-linked list (SLL). In C, a data type known as struct, could act as SLL. SLL make objects to be iterated by recursion.

|  |
| --- |
|  |
| class Node:  def \_\_init\_\_(self, name, node=None):  self.name = name  self.next = node |

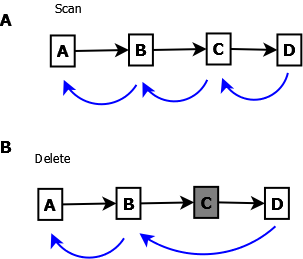


Figure . Recursively access each node from the first node to the end.

Figure 9 reveals that pattern of recursive accessing SLL acts the same as accessing a "stack" - "First IN, Last OUT". Access starts from the first node and ends at the end node. Recursion return from the most inner function and finally return the first node.

There are similar writing styles. style 3 has some abundant codes compared with the other two. I recommend to remember style 3 because those code could be modified for more complicated functions namely, insertion or deletion.

|  |
| --- |
| output:  A B C D None |
| #style 1:  class Chain:  def scan(self, node):  if node is not None:  print(node.name, end="\t")  self.scan(node.next)  #create a chain  chain = Node("A", Node("B", Node("C", Node("D"))))  #scan a chain  res = Chain().scan(chain)  print(res) |
| #style 2:  class Chain:  def scan(self, node):  if node is None:  return None  else:  print(node.name, end="\t")  self.scan(node.next)  #  chain = Node("A", Node("B", Node("C", Node("D"))))  res = Chain().scan(chain) |
| #style 3:  class Chain:  def scan(self, node):  if node is None:  return node  else:  print(node.name, end="\t")  **node.next = self.scan(node.next)**  **return node**  #  chain = Node("A", Node("B", Node("C", Node("D"))))  res = Chain().scan(chain)  print(res.name) |

Moreover, closely pay attention to the approach of return statement in recursive functions. It is ok to return the first node, last node or one of any other nodes, or None.

|  |  |
| --- | --- |
|  |  |
| output:None | class Chain:  def scan(self, node):  if node is None:  **return None**  else:  **return self.scan(node.next)**  chain = Node("A", Node("B", Node("C", Node("D"))))  res = Chain().scan(chain)  print(res) |
| output: D  return the target node | class Chain:  def scan(self, node):  if node.next is None:  **return node**  else:  **return self.scan(node.next)**  chain = Node("A", Node("B", Node("C", Node("D"))))  res = Chain().scan(chain)  print(res.name) |
| output: A  return the first node. The approach is useful when some updates on other nodes are needed. | class Chain:  def scan(self, node):  if node is None:  pass  else:  **self.scan(node.next)**  **return node**  chain = Node("A", Node("B", Node("C", Node("D"))))  res = Chain().scan(chain)  print(res.name) |

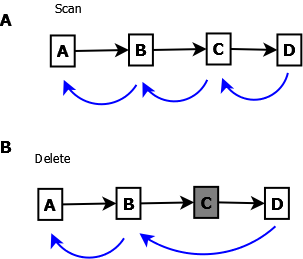


Figure . Delete a node by return the next node in SSL.

The next, let's consider how to delete one node from the chain (Figure 10).

|  |
| --- |
| output:  A B D |
| class Chain:  def scan(self, node):  if node is None:  return None  else:  print(node.name, end="\t")  return self.scan(node.next)  def delete(self, node, val):  if node is None:  return None  elif node.name == val:  **return node.next**  else:  **node.next = self.delete(node.next, val)**  **return node**  chain = Node("A", Node("B", Node("C", Node("D"))))  res=Chain().delete(chain, "C")  Chain().scan(res) |

The function below would update values of each node, and then return starting node.

|  |
| --- |
| output:  A-B-C-D- |
| class Chain:  def scan(self, node):  if node is None:  return None  else:  print(node.name, end="")  return self.scan(node.next)  def update(self, node, val):  if node is None:  return None  else:  **node.name += val**  **node.next = self.update(node.next, val)**  **return node**  chain = Node("A", Node("B", Node("C", Node("D"))))  res=Chain().update(chain, "-")  Chain().scan(res) |

# matrix and distance

## general distance

Given two collections *p* and *q* with *n* elements, the distance is

This formula could be used for identification of two permuted strings. For example, "abc", "cba", and "bac" *etc*. are permuted. It is ok to sort them before comparison in order to identify if they are permutation. but that approach would consume two much memory and time for comparing long permuted strings. Calculation of distance would solve this issue. shows the distance of any two of them are the same.

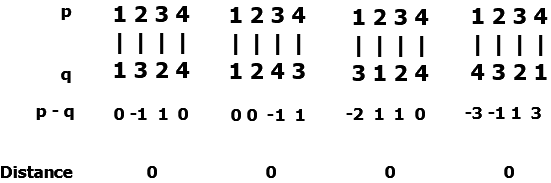


Figure . Calculate general distance of two permuted collections.

|  |
| --- |
| output:  Permutated strings:  a-a=-3  b-b=-4  c-c=-3  d-d=0  distance is 0.  None permutated strings:  d-d=1  c-c=1  b-b=1  a-a=1  distance is 1. |
| def string\_distance(s1, s2):  d = 0  for p, q in zip(s1, s2):  d+= ord(p) - ord(q)  print(f"\t{p}-{p}={d}")  print(f"distance is {d}.")  print("Permutated strings:")  string\_distance("abcd", "dcba")  print("None permutated strings:")  string\_distance("dcba", "ccba") |

Moreover, we could modify this approach by calculating relative distance. Given any of string with n characters, we could calculate the distance between this string and a fixed string with n 'A'. Showed as the functions below, the string "abc" has 6 permutations. All of their relative distance is 99.

|  |
| --- |
| output:  abc, distance=99  acb, distance=99  bac, distance=99  bca, distance=99  cba, distance=99  cab, distance=99 |
| def cal\_distance(s):  if s == "":  return 0  d = ord(s[0]) - ord('A')  return d + cal\_distance(s[1:])  def recur\_seq(s, L, R):  if L==R:  ns = "".join(s)  d = cal\_distance(ns)  print(f"{ns}, distance={d}")  else:  for i in range(L, R+1):  s[L], s[i] = s[i], s[L]  recur\_seq(s, L+1, R)  s[L], s[i] = s[i], s[L]  def permute\_string(s):  L, R = 0, len(s)-1  recur\_seq(list(s), L, R)  #  permute\_string("abc") |

## Euclidian distance

If p and q are two points on a line, the distance of p and q is

if there are two space p and q with n-dimensional, the distance of p and q is

Euclidian distance for compare two permuted strings.

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

# Binary tree

Strength of storing data in array:

* Elements of array is in order.
* support random access and iteration.

weakness of storing data into array:

* Can't delete, insert elements. Array type is static type in C/C++ or Java.

Alternatives for array data type. Hash table is one option. In Python, that is dictionary type. hash table allows deletion and insertion, support to access value by key. but hash table doesn't maintain data pairs in order.

Binary tree is very good option. Binary tree is node-based data structure. Any one of nodes in binary tree has no, or one or two child nodes.

## create a tree

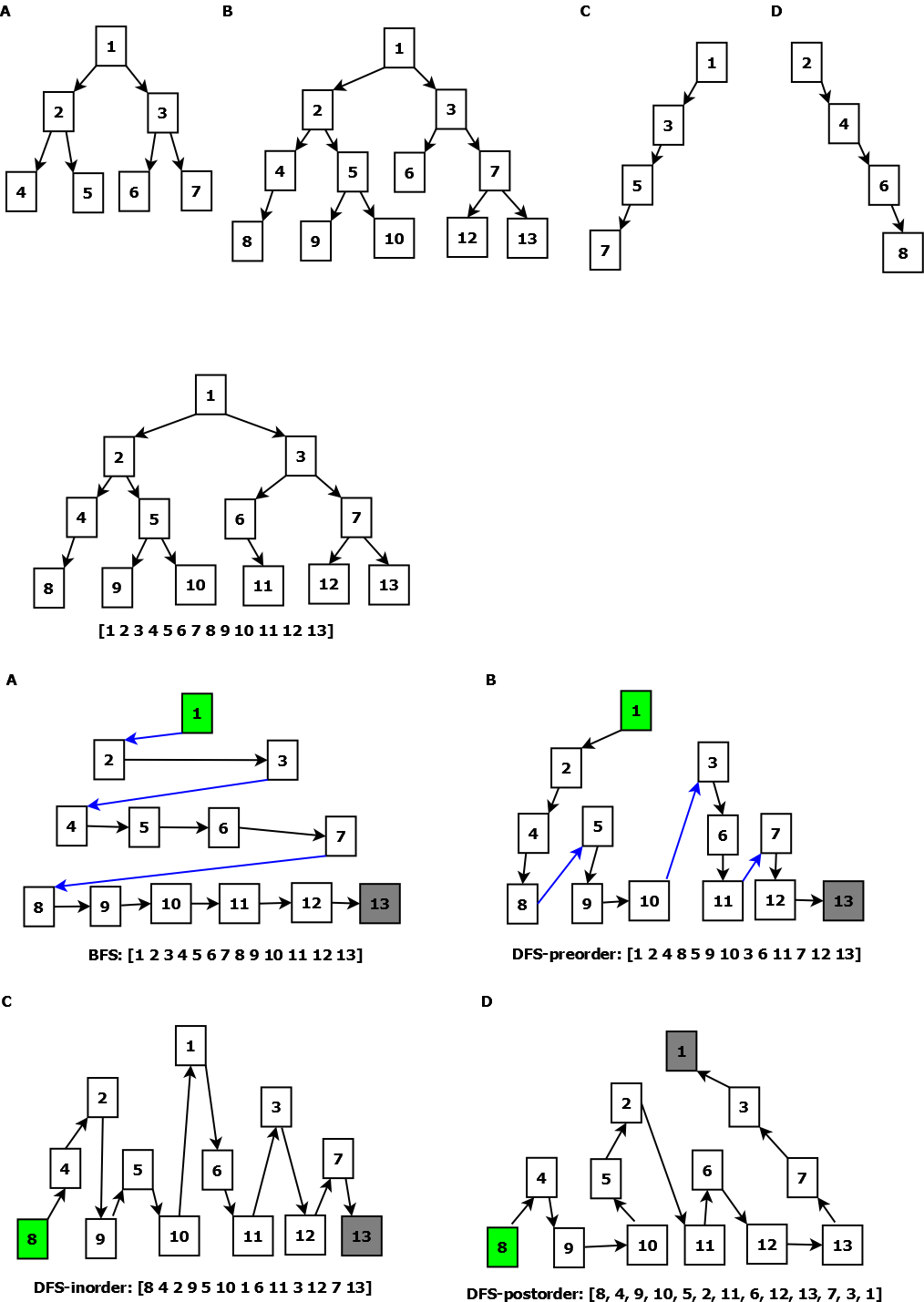


Figure . Types of binary tree. A. full binary tree. B. complete binary tree. C. left-leaves binary tree. D. right-leaves binary tree.

|  |
| --- |
|  |
| class TreeNode:  def \_\_init\_\_(self, val=0, left=None, right=None):  self.value = val  self.left = left  self.right = right |

|  |
| --- |
|  |
| TREE1 = TreeNode(1, TreeNode(2, TreeNode(4, TreeNode(8)), \  TreeNode(5, TreeNode(9), TreeNode(10))), \  TreeNode(3, TreeNode(6, None, TreeNode(11)), \  TreeNode(7, TreeNode(12), TreeNode(13)))  )  TREE2 = TreeNode(1, TreeNode(3, TreeNode(5, TreeNode(7))))  TREE3 = TreeNode(2, None, TreeNode(4, None, \  TreeNode(6, None, TreeNode(8)))) |

## tree traversal

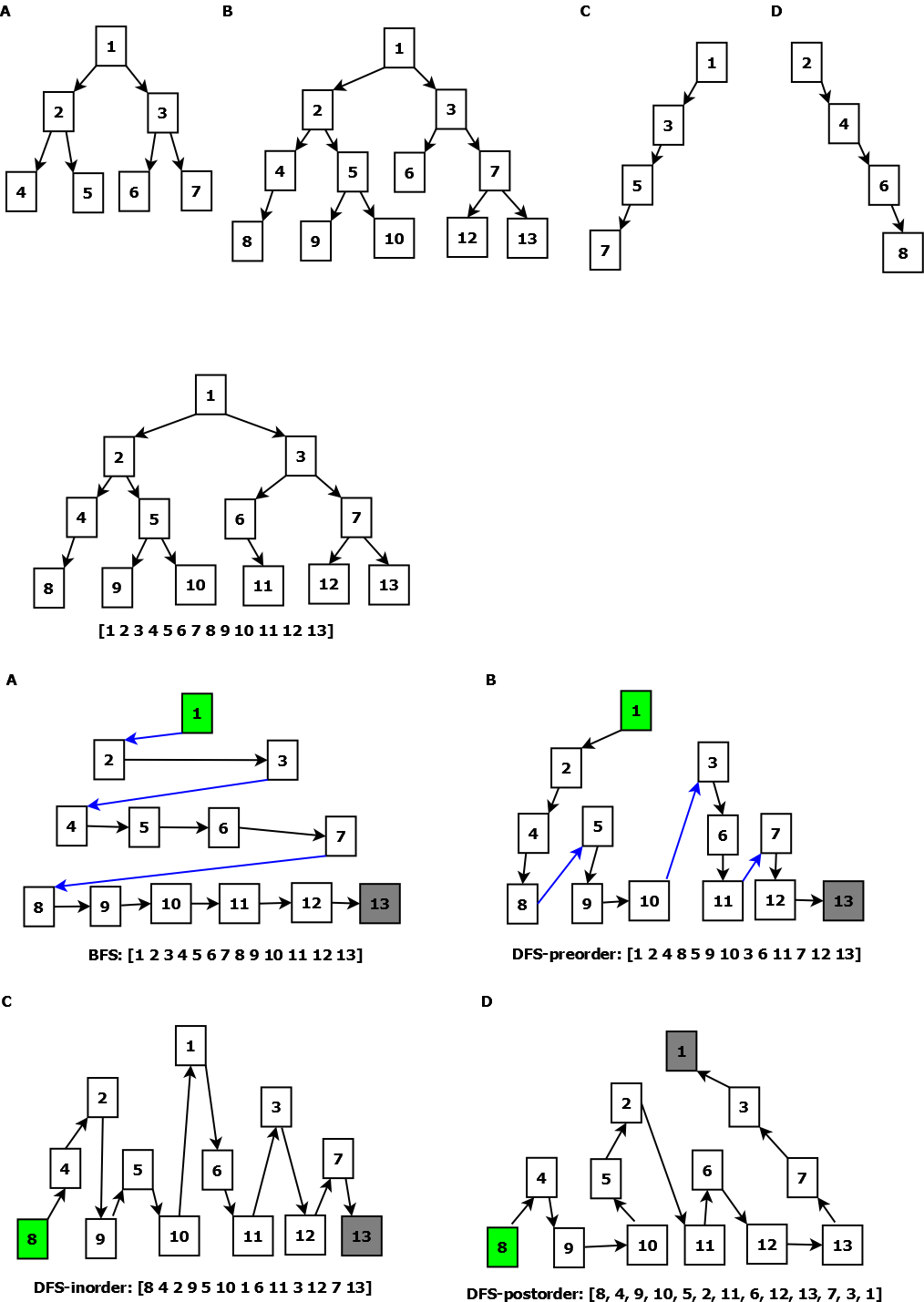


Figure . Example of binary tree.

level order traversal (breadth first search)

|  |
| --- |
|  |
| def scan\_BFS(self, root):  if not isinstance(root, TreeNode):  return []  values = []  pool = [root]  while(pool):  node = pool.pop(0)  if node is not None:  values.append(node.value)  pool.append(getattr(node, 'left', None))  pool.append(getattr(node, 'right', None))  else:  values.append(None)  values = self.removeTailNone(values)  return values  def removeTailNone(self, values: List) -> List:  if len(values)>0 and values[-1] is None:  return self.removeTailNone(values[:-1])  else:  return values |

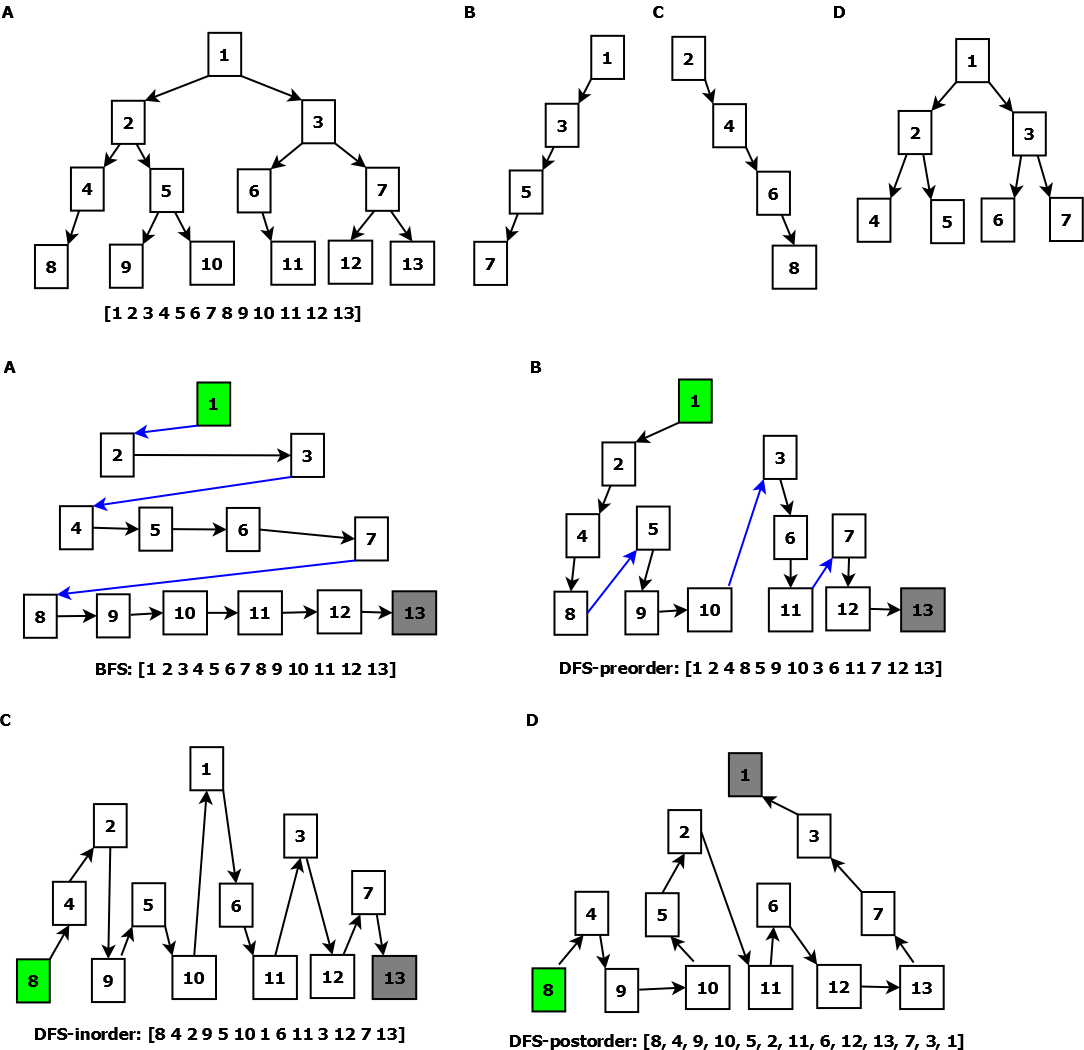


Figure . Tree traversal. A. Breadth First Search. B. Depth First Search in pre-order (root->left->right). C. Depth First Search in in-order (left->root->right). Depth First Search in post-order (left->right->root).

|  |
| --- |
|  |
| def scan\_DFS\_preorder(self, root):  if not isinstance(root, TreeNode):  return []  return [root.value]+ self.scan\_DFS\_preorder(root.left) \  + self.scan\_DFS\_preorder(root.right) |
| def scan\_DFS\_inorder(self, root):  if not isinstance(root, TreeNode):  return []  return self.scan\_DFS\_inorder(root.left) + [root.value]\  + self.scan\_DFS\_inorder(root.right) |
| def scan\_DFS\_postorder(self, root):  if not isinstance(root, TreeNode):  return []  return self.scan\_DFS\_postorder(root.left) + \  self.scan\_DFS\_postorder(root.right) + [root.value] |

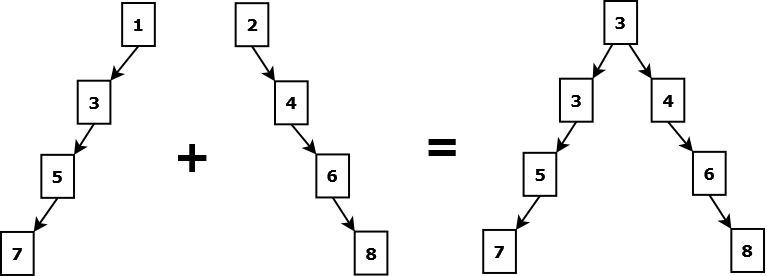


Figure . Merge two binary tree.

Figure 13. shows how to merge two binary tree into one tree. Plus vales if both of them exist.

|  |
| --- |
|  |
| def mergeTrees(self, t1: TreeNode, t2: TreeNode):  if t1 is None and t2 is None: return None  if t1 is None: return t2  if t2 is None: return t1  t = TreeNode(t1.value + t2.value);  t.left = self.mergeTrees(t1.left, t2.left);  t.right = self.mergeTrees(t1.right, t2.right);  return t |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## binary search tree

For storage of ordered data, there is binary search tree (BST). Binary search tree is one kind of binary tree. Here is the rules, given a certain node,

* That node may have no, one or at most two children nodes.
* Left subtree of that node contains those nodes of which all values are less than the value of that node.
* Right subtree of that node contains those nodes of which all values are greater than the value of that node.

Therefore, if a certain node has two children nodes, the two nodes are sibling node to each other. The value of left node is always less than that of the node, and the value of right node is always greater than that of the node.

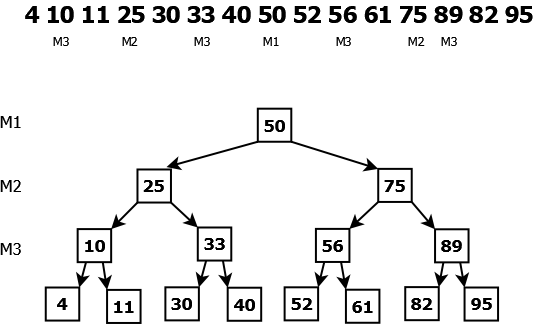


Figure .Binary search Tree following middle point rule.

Notes:

* BST could be full binary tree, or complete binary tree, or just binary tree.
* BST stores a sorted collection. BST is useful for searching collections by values.
* Values of nodes in BST should be unique. The attribute of "TreeNode.count" allows duplicate values, which is useful for searching highly skewed data.
* The rule of middle points is allowed in BST showed as in Figure 14. Parent nodes including root node rather than end nodes are actually middle points in a sorted collection. Adjacent elements of those middle points in collections are end nodes in BST. This rule balances nodes attributions of BST, which makes BST to be symmetric. Of course, the rule is not required for BST.

|  |
| --- |
|  |
| class TreeNode:  def \_\_init\_\_(self, val, left=None, right=None):  self.value = val  self.count = 0  self.left = left  self.right = right |
| TREE1 = TreeNode(50,  TreeNode(25, TreeNode(10, TreeNode(4), TreeNode(11)), \  TreeNode(33, TreeNode(30), TreeNode(40))),  TreeNode(75, TreeNode(56, TreeNode(52), TreeNode(61)), \  TreeNode(89, TreeNode(82)))  ) |

## search tree

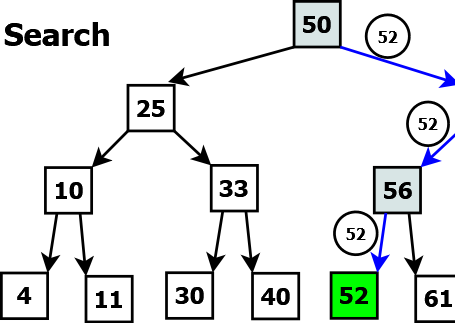


Figure . Search BST using breadth-first search.

|  |
| --- |
|  |
| def search(self, root, val):  if not isinstance(root, TreeNode):  return False  else:  if root.value == val:  return True  elif root.value > val:  return self.search(root.left, val)  return self.search(root.right, val) |

The above function return True or False. but how about return the index in the sorted collection if something is detected. Given a sorted collection with n elements, the kth element should be returned.

|  |
| --- |
|  |
|  |

There is indexed BST. New fields known as left\_size and right\_size are added into class TreeNode. "left\_size" and "right\_size" are the number of nodes in its left-subtree plus 1 and right-subtree plus 1, respectively.

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## insert node

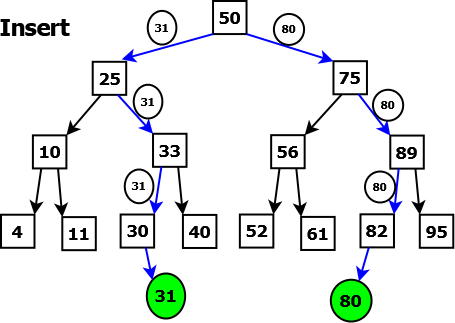


Figure . Insert nodes into BST.

For insertion, the new node would always be placed as end nodes showed as in Figure 16.

|  |
| --- |
|  |
| def insert(self, root, val):  if not isinstance(root, TreeNode):  return TreeNode(val)  else:  if val == root.value:  root.count += 1  return root  elif val < root.value:  root.left = self.insert(root.left, val)  else:  root.right = self.insert(root.right, val)  return root |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## delete node

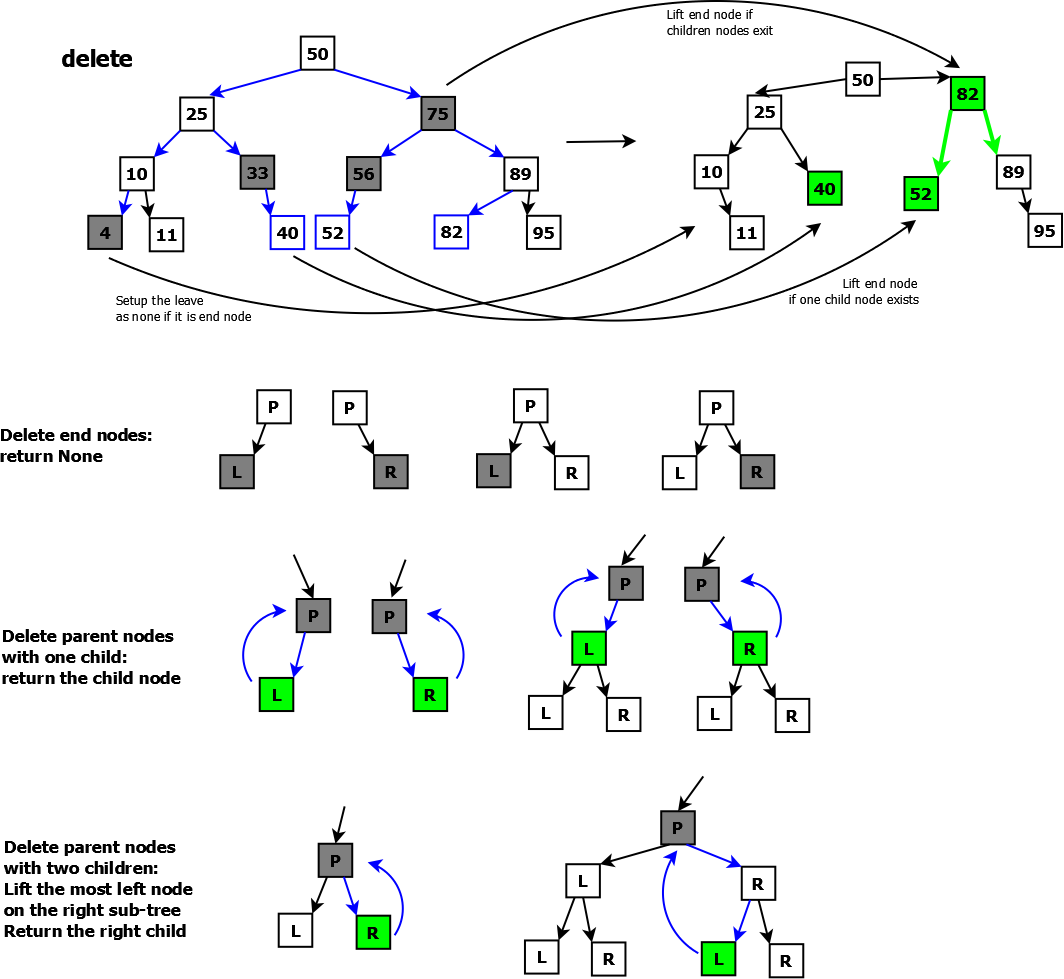


Figure . Consider possible deletion of a node from BST.

Figure 16. shows all possibilities of deletion in BST.

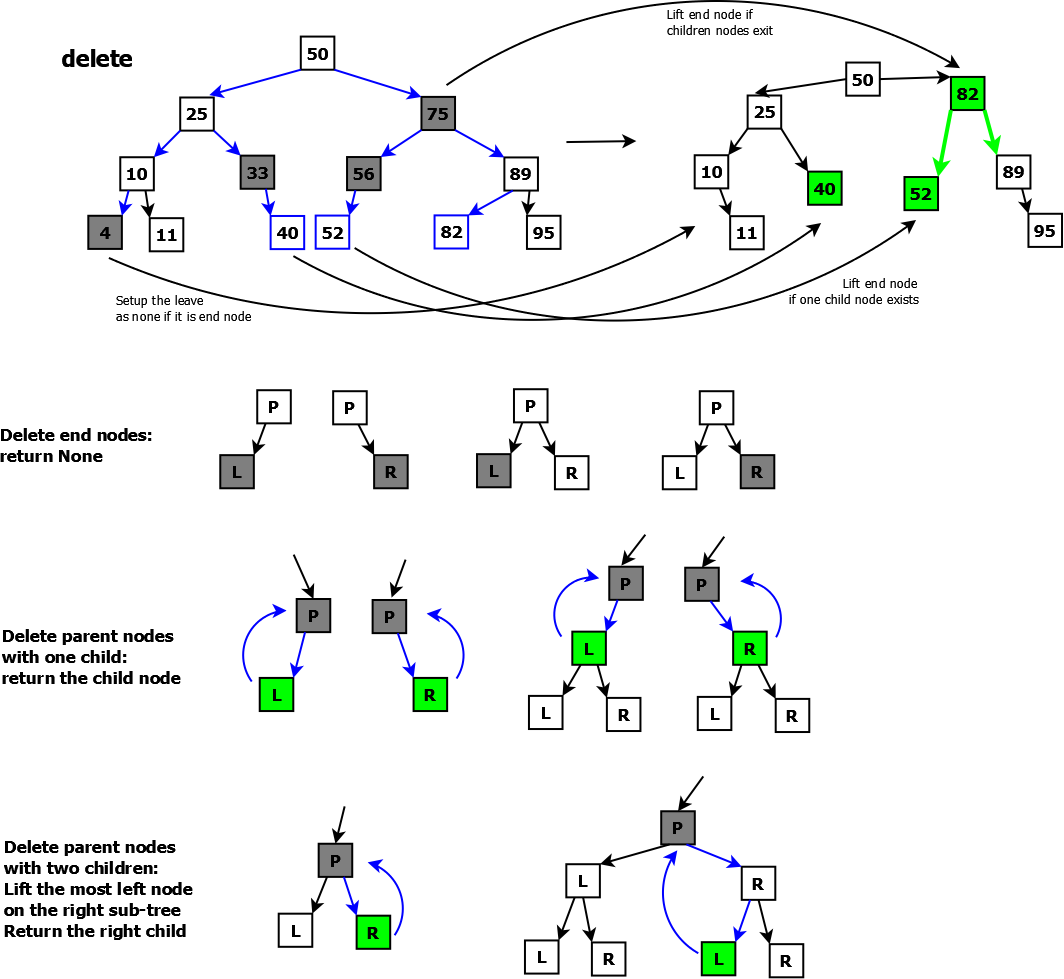


Figure . Delete nodes from an example BST.

|  |
| --- |
|  |
| class BinarySearchTree:  def delete(self, node, val):  #search the node that should be deleted  if node is None:  return None  elif val < node.value:  node.left = self.delete(node.left, val)  return node  elif val > node.value:  node.right = self.delete(node.right, val)  return node  elif val == node.value:  #either or both of left-right is none  if node.left is None:  return node.right  if node.right is None:  return node.left  #Both of left and right exist  node.right = self.liftValue(node.right, node)  return node    def liftValue(self, nodeRight, nodeDel):  #walk through the most left path  if nodeRight.left:  nodeRight.left = self.liftValue(nodeRight.left, nodeDel)  return nodeRight  #return the most left end node  else:  nodeDel.value = nodeRight.value  return nodeRight.right |

|  |
| --- |
|  |
|  |

## retrieval

The abbreviation of "retrieval" is "trie", which is pronounced as "try". "trie" is tree consisting of nodes. Differing from binary tree, a trie-node can have any number of children nodes. The node is represented by hash table. In Python, a "trie" could be a nested dictionary. A "trie" could be converted to json format for data communication cross various platform.

Hash table could store words or some strings, which could be decompose multiple letters or digits. Retrieval could be used for autocomplete, autocorrect, or validate IP addresses or phone numbers.

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

# Bit manipulation

convert decimal integer to binary:

{4}10 = {100}2{3}10 = {011}2

{13}10 = {1101}2

Features of binary form:

* only consist of 1 or 0.
* even and odd numbers end with 0 and 1, respectively.

Given integer x (x>0), there is an expression "x & (x-1)". Consider some rules showed as the below:

* The product y is always even number because either x or x-1 is even.
* The number of ones of is less one than that of x. for example: 15 = 1111, 15&14=14=1110. number of ones of 15 and 14 is 4 and 3, respectively.
* If y is equal to 0, x would be the power f 2. for example: 4&(3)=0
* Recursively execute the expression "x&(x-1)" would update all ones to zeros from the most right side to the left. Finally that returns zero.

|  |
| --- |
| output:  adjacent -> x&(x-1)  1-0: 0b1-0b0 -> 0: 0b0  2-1: 0b10-0b1 -> 0: 0b0  3-2: 0b11-0b10 -> 2: 0b10  **4-3: 0b100-0b11 -> 0: 0b0**  5-4: 0b101-0b100 -> 4: 0b100  6-5: 0b110-0b101 -> 4: 0b100  7-6: 0b111-0b110 -> 6: 0b110  **8-7: 0b1000-0b111 -> 0: 0b0**  9-8: 0b1001-0b1000 -> 8: 0b1000  10-9: 0b1010-0b1001 -> 8: 0b1000  11-10: 0b1011-0b1010 -> 10: 0b1010  12-11: 0b1100-0b1011 -> 8: 0b1000  13-12: 0b1101-0b1100 -> 12: 0b1100  14-13: 0b1110-0b1101 -> 12: 0b1100  15-14: 0b1111-0b1110 -> 14: 0b1110  **16-15: 0b10000-0b1111 -> 0: 0b0**  17-16: 0b10001-0b10000 -> 16: 0b10000  18-17: 0b10010-0b10001 -> 16: 0b10000  19-18: 0b10011-0b10010 -> 18: 0b10010  20-19: 0b10100-0b10011 -> 16: 0b10000 |
| def twoAdjacents(start, end):  if start > end:  return None  else:  adj = start - 1  p = start & adj  print(f"{start}-{adj}: {bin(start)}-{bin(adj)} -> {p}: {bin(p)}")  twoAdjacents(start+1, end)  print(f"adjacent -> x&(x-1)")  twoAdjacents(1,20) |

remember the usage of bitwise operations, given a binary number x

Get the last digits: x&1

{23}10 ={10111}2, 23&1 = 1

{10}10 ={1010}2, 10&1 = 0

Remove the last digits: x>>1

{23}10 ={10111}2, 23>>1 = {1011}2 = 11

{10}10 ={1010}2, 10>>1 = {101}2 = 5

Append zero to bits: x<<1

{23}10 ={10111}2, 23<<1 = {101110}2 = 46

{10}10 ={1010}2, 10<<1 = {10100}2 = 20

Add 1 on the first of the bits of x: (1<<(len(bin(x))-2)) + x

{23}10 ={10111}2 + 1<<5 = {100000}2 = {110111}2

|  |
| --- |
|  |
|  |

Concatenate two bits:

{23}10 ={10111}2 + {10}10 ={1010}2 -> {101111010}2

|  |
| --- |
| output:  0b10111 0b1010 378 0b101111010 |
| x, y = 23, 10  r = (x<<(len(bin(y))-2))+10  print(bin(x), bin(y), r, bin(r)) |

|  |
| --- |
|  |
|  |

# Graph

## graph representation

adjacency matrix

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

## traverse matrix

Given a matrix with m x n. rows and columns are m and n, respectively.

BSF traversal

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |

|  |
| --- |
|  |
|  |