Basic Data Structrues

Yanhua Huang

Mar 2018

Here are some notes for understanding basic data structures with Python examples.

Array

An **array** stores a collection of elements in contiguous memory locations. By linear addressing, it supports random access (more precisely called direct access). However, the insert and delete are inefficient because of elements moving for keeping contiguous memory. Here is a insert example to show the dynamic allocation.

```python
import sys
from matplotlib import pyplot as plt
n = 10000
obj_list = []
obj_tuple = tuple([])
for _ in range(n):
    obj_list.append(sys.getsizeof(obj_list))
    obj_tuple += (sys.getsizeof(obj_tuple), )
plt.plot(range(n), obj_list, '-b', label='dynamic allocation')
plt.plot(range(n), obj_tuple, '-r', label='static allocation')
plt.legend()
plt.show()
```

Linked List

A **linked list** consists of a collection of nodes that together represent a sequence. The most basic form is **single linked list** where each node contains data and a pointer that refers to the next node or null for the last node. Complex forms contain doubly linked list for traversal in either direction, and circular linked list for circle structure puzzles like Josephus problem. Comparing with array, it can use separate memory and is efficient for insertion or removal, but occupies more memory and do not allow random access.

Here is a Least Recently Used cache strategy implemented by a linked list, where we only consider the number of cached elements.

```
class Cache(object):
    def __init__(self, size=100):
        self._storage = []
        self._maxsize = size

    def add(self, obj):
        address = id(obj) # get the address
        index = 0
        for element in self._storage:
            if element == address: # object is already in cache, move
                to the head
                self._storage.pop(index)
                self._storage.insert(address, 0)
                break
            index += 1
        else: # object is not in cache
            if len(self._storage) < self._maxsize: # cache is not
                full, insert to the head
                self._storage.insert(address, 0)
            else: # cache is full, replace the tail
                self._storage[-1] = address
```

Skip List

A **skip list** uses a linked list with multi-level indexes to store ordered sequences, which allows search and insertion in $\mathcal{O}(logn)$ time. When inserting a new element, the skip list maintains the balance by some carefully designed random function. Skip list is a crucial data structure in sorted sets of Redis for efficient interval searching.

Stack

A **stack** is a linear data structure with two principal operations: push and pop, where push adds an element and pop removes the most recently added element. Both push and pop occur only at one end of the stack, referred to as the top of the stack. The stack can be implemented by array or linked list. It is a useful data structure for evaluating expressions such as algebraic expressions or most generally function calls. Here is an example of identifying illegal brackets, where we only consider square brackets.

```
def is_illegal_brackets(expression):
    stack = [] # the tail of the list is the top of the stack
    for c in expression:
        if c == '[': # push to the stack
            stack.append(c)
        else:
            if stack[-1] == '[': # match [
                stack.pop()
            else: # mismatch
                return True
    return bool(stack) # if stack is empty then return False
is_illegal_brackets('[[]]') # return False
is_illegal_brackets('[[]') # return True
is_illegal_brackets('][]') # return True
```

Queue

A **queue** is a linear data structure with two principal operations: enqueue and dequeue, where enqueue adds an element and dequeue removes the most earliest added element. Enqueue and dequeue occur at two different ends of the queue, referred to as the tail and head, respectively. For resources limit situation, the queue can be used to control the priority.

Tree

A **tree** is a hierarchical data structure, defined recursively as a collection of nodes. Node's height means the number of edges on the longest path between it and a descendant leaf. Node's depth means the distance between it and the root. Node's level is equal to its depth plus 1. The tree's height is the height of the root node. The tree is always implemented like linked list, but complete binary tree can be implemented by array.

Binary search trees satisfy that nodes in the left subtree $<$ current node $<$ nodes in the right subtree while duplicate data can be implemented by storing multiple data in a single node or viewing equal as greater. Insertion can be implemented by adding a new element to the searched leaf node while deletion can be implemented by first moving data in any leaf node to the current node and then swapping data recursively.

Heap

A **heap** is a complete binary tree where each node must be greater or less than its descendants. For node with index i, indexes of its children are $(i << 1) + 1$ and $(i << 1) + 2$, and index of its father is $i >> 1$. Insertion can be implemented by first appending to the tail and swapping recursively while deletion can be implemented by first moving the tail to the current node and then swapping recursively. Sorting by heap needs pop the root and rebuild the heap repeatly, which requires more data swap and is not friendly to cpu cache. Heap is useful when calculating quantile of streaming sequence by maintaining a max heap and a min heap, where insertion only moves log(n) elements while an ordered array moves n / 2 elements after binary search. Another application is merging ordered sequences by maintaining the first elements in sequences in a heap.

Sort

Sort algorithm with $\mathcal{O}(1)$ spatial complexity is called sorted in place. The times of moving and comparison influence the time complexity of a sort algorithm. A sort algorithm is stable if and only if it maintains the order of the same elements in the raw sequence.

```python
def swap(seq, i, j):
    tmp = seq[i]
    seq[i] = seq[j]
    seq[j] = tmp


def bubble_sort(seq):
    n = len(seq)
    for done in range(n): # done interval is in tail
        flag = False
        for i in range(1, n - done):
            if seq[i - 1] > seq[i]: # swap with three operations
                swap(seq, i - 1, i)
                flag = True
        if not flag:
            break
    return seq


def insertion_sort(seq):
    n = len(seq)
    for done in range(1, n): # done interval is in head
        val = seq[done]
        for i in range(done)[::-1]: # swap with only one operation
            if seq[i] > val:
                seq[i + 1] = seq[i]
            else:
                seq[i + 1] = val
                break
        else:
            seq[0] = val
    return seq
```

```python
def selection_sort(seq):
    n = len(seq)
    for done in range(n): # done interval is in head
        index = done
        value = seq[done]
        for i in range(done + 1, n):
            if seq[i] < value:
                value = seq[i]
                index = i
        swap(seq, done, index) # swap is not stable
    return seq


def quick_sort1(seq, start=0, end=None): # recursion with build-in
     function stack
    n = len(seq)
    if n <= 1:
        return seq
    if end is None:
        end = n
    val = seq[end - 1]
    index = start
    for i in range(start, end - 1):
        if seq[i] < val:
            swap(seq, index, i)
            index += 1
    swap(seq, index, end - 1)
    if index - start > 1:
        quick_sort1(seq, start, index)
    if end - index - 1 > 1:
        quick_sort1(seq, index + 1, end)
    return seq


def quick_sort2(seq): # recursion with manual function stack
    n = len(seq)
    if n <= 1:
        return seq
    stack = [(0, n)]
    while stack:
        start, end = stack.pop()
        val = seq[end - 1]
        index = start
        for i in range(start, end - 1):
            if seq[i] < val:
                swap(seq, index, i)
                index += 1
        swap(seq, index, end - 1)
        if index - start > 1:
            stack.append((start, index))
        if end - index - 1 > 1:
            stack.append((index + 1, end))
    return seq


def merge_sort(seq):
    n = len(seq)
    split = 1
    while split < n:
        res = [] # not sorted in place
        step = split * 2
        for i in range(0, n, step):
            offset1 = i
            end1 = i + split
            offset2 = i + split
            end2 = min(n, i + step)
            while offset1 < end1 and offset2 < end2:
                if seq[offset1] < seq[offset2]:
                    res.append(seq[offset1])
                    offset1 += 1
                else:
                    res.append(seq[offset2])
                    offset2 += 1
            res += seq[offset1:end1] + seq[offset2:end2]
        seq = res
        split *= 2
    return seq
```