Full CSC111 Notes

Tuesday, 8 April 2025 9:24 PM

Linked List

```
class LinkedList:

    _first : Optional[_Node]

    def __init__(self) -> None:
        self._first = None
```

```
class _Node:
    item : Any
    next : Optional[_Node] = None
```

- __future__: allows call of itself(class) in its attributes

- Code Template: (1) Not None (2) Not at index i

(A)

```
curr = self._first
curr_index = 0
while curr is not None:
    ... curr.item...
    curr = curr.next
    curr_index = 0
```

(B)

```
curr = self._first
curr_index = 0

while not (curr is None or curr_index == i)
    curr = curr.next
    curr_index = curr_index + 1

if curr is None:
    raise IndexError
else:
    return curr.item
```

```
linky[2] = linky.__getitem__(2)
```

- Mutation
  - Append:

    Case 1: Empty List

    Case 2: Non-empty list

```
if self._first is None :
    self._first = _Node(item)
else:
    curr = self._first
    while curr.next is not None
        curr = curr.next

    curr.next = _Node(item)
```

Running-time analysis:

Linked List:

Indexing = $\Theta(i)$

Inserting = $\Theta(i)$

Removing = $\Theta(i)$

} Finding index i-1

- Index-Based Mutation
  - access node i-1 to insert at i.

```
def insert(self, i : int, item : Any) -> None:
    new_node = _Node(item)
    if i == 0 :
        new_node.next = self._first
        self._first = new_node

    curr = self._first
    curr_index = 0
    while not (curr is None or curr_index == i-1)
        curr = curr.next
        curr_index = curr_index + 1

    if curr is None:
        raise IndexError
    else:
        new_node = _Node(item)
        new_node.next = curr.next
        curr.next = new_node
```

Nested List (unpredictable list in list)

```
def sum_nested(nested_list : int | list) -> int:
    if isinstance(nested_list, int):
        return nested_list
```

Base Case: nested_list is integer

```
    else:
        for sublist in nested_list:
            sum_so_far = 0
            sum_so_far += sum_nested(sublist)
 # OR     sum(sum_nested(sublist) for sublist in nested_list)
    return sum_so_far
```

Call itself on each item in the nested_list

Recu

✗ Depth = the number of nested list enclosing that value.

```
 1    2 3
[1, [[2, 3], 4]]
 ↑     ↑
depth=1  depth=3
```

```
• def flatten(nested_list: int | list) -> list[int]:
      if isinstance(nested_list, int):
          return [nested_list]
      else:
          result_so_far = []
          for sublist in nested_list:
              result_so_far.extend(flatten(sublist))
          return result_so_far
```

```
• def nested_list_contains(nested_list: int | list, item: int) -> bool:
      if isinstance(nested_list, int):
          return nested_list == item
      else:
          any(nested_list_contains(sublist) for sublist in nested_list)
```

[1, [[2, 3], 4]]

```
def first_at_depth(nested_list: int | list) -> in
    if isinstance(nested_list, int):
        if d == 0:
            return nested_list
        else:
            return None
    else:
        if d == 0:
            return None
        else:
            for sublist in nested_list:
                item = first_at_depth(sublist, d-1
                if item is not None:
                    return item
        return None
```

("1" has 0

([1] has 1 d

e.g.

[2,3],

Recursive List

```
_first : Optional[Any]
_rest : Optional[RecursiveList]
```

} Empty Recursin List
- first = None
- rest = None

```
def sum(self) -> int:
    if self._first is None:
        return 0
    else:
        return self._first + self._rest.sum()
```

R = [(int), [R]]
↑ first ↑ rest

```
[1, [2, []]]
        ↑
      None => 0.
        ↓
     2 + 0 = 2.
   ↓
1 + 2 = 3.
```

Trees.

• Size : number of nodes

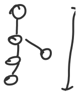
```
class Tree:
    _root: Optional [Any]
    _subtrees: list [Tree]
    def __init__(self, root: Optional[Any], _subtrees: list[Tree]) -> None:
        self._root = root
        self._subtrees = subtrees
    def is_empty(self) -> bool:
        return self._root is None
```

Template:

```
def method():
    if self.is_empty():
        ...
    elif self._subtree = []:
        ...
    else:
        ...
        for subtree in self._subtrees:
            ... subtree.method()...
        ...
```

```
def average(self) -> float:
    if self.is_empty:
        return 0.0
    else:
        sum_items, num_items = self._average_helper()
def _average_helper(self) -> tuple[int,int]
    if self.is_empty():
        return (0,0)
    elif self._subtrees == []:
        return (self._root, 1)
    else:
        sum_so_far = self._root
        size_so_far = 1
        for subtree in self._subtrees:
            sub_sum, sub_size = subtree._average_helper()
            sum_so_far += sub_sum
            size_so_far += sub_size
        return sum_so_far, size_so_far
```

- leaf: a value with no subtree
- internal value: a node that is not a leaf
- height: longest path from root to its leaf.



- children: directly under that node
- descendants: every value under that node
- parent/ancestor, similar.

```
def __len__(self):
    if self.is_empty():
        return 0
    else:
        return 1 + sum(subtree.__len__() for sub
```

Remove (Mutation)

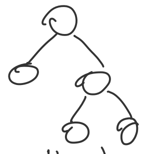
```
def remove(self, item: Any) -> bool:
    if self.is_empty():          # Can't find!
        return False
    elif self._root == item:           # Found!
        self._delete_root 1/2()
        return True
    else:
        for subtree in self._subtrees:
            if subtree.remove(item):   # Found!
                return Tree
        return False
```

i

de

def

```
# Strategy 1: Promote the last subtree
def _delete_root_1():
    if self.subtrees == []: # This is leaf
        self._root = None
    else:
        last_subtree = self._subtrees.pop
        self._root = last_subtree._root
        self._subtrees.extend(last_subtree._subtrees)
```



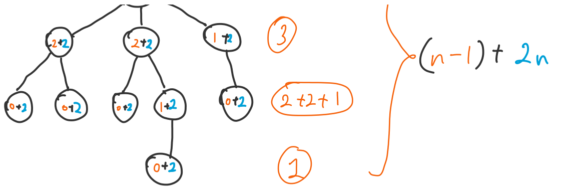
Running-time Analysis

1. Find number of recursive calls (No. of subtrees k)
2. Find number of non-recursive calls (No. of constant calls)

```
def __len__(self):
    if self.is_empty():     ] 1
        return 0
    else:
        sum_so_far = 1     } 1 step
        for subtree in self._subtrees:
            sum_so_far += subtree.__len__()     ] k steps
        return sum_so_far } 1 step
```

Each call for each node, have n nodes
= k+2 steps

Recursive Call diagram



$(n-1) + 2n$

Binary Search Trees

- Multiset : abstract data type, extension of set ADT that alows duplicate

```
class BinarySearchTree :

    - root : Optional[Any]
    - left : Optional[BinarySearchTree]
    - right : Optional[BinarySearchTree]

    def __init__(self, root: Optional[Any]):
        if root is None:
            self._root  = None
            self._left  = None
            self._right = None

        else:
            self._root = root
            self._left = BinarySearchTree(None)
            self._right = BinarySearchTree(None)
```

- None if BST is empty.
- BST property:

If self._root is not None
- self._left <= self._root
- self._right >= self._root

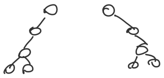
```
def __contains__(self, item: Any) -> bool:
    if self.is_empty():
        return False
    elif self._root == item:
        return True
    elif item < self._root:
        return self._left.__contains__(item)
    else:
        return self._right.__contains__(item)
```

```
def insert(self, item: Any) -> None:
    if self.is_empty():  # Base Case: find empty
        self._root = item
        self._left, self._right = BinarySearchTree(None), BinarySearchTree(None)
    else:
        if item <= self._root:
            self._left.insert(item)
        else:
            self._right.insert(item)
```

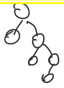
- Deletion of root
  1. Replace with left's greatest
  2. Replace with right's smallest

```
def remove(self, item: Any) -> None:
    if self.is_empty():
        pass
    elif self._root == item:
        self._delete_root()
    elif item < self._root:
        self._left.remove(item)
    else:
        self._right.remove(item)
```

```
def _delete_root(self) -> None:
    if self._left.is_empty() and self._right.is_empty():  # Leaf.
        self._root = None
        self._left = None
        self._right = None

    elif self._left.is_empty():
        self._root, self._left, self._right =
        self._right._root, self._right._left, self._right._right

    elif self._right.is_empty():
        self._root, self._left, self._right =
        self._left._root, self._left._left, self._left._right

    else:
        self._root = self._right.extract_max()
```



```
def extract_max(self):
    if self._right.is_empty():
        max_item = self._root
        self._root = self._left._root
        self._left, self._right = self._left._left, self._l
        return max_item
    else:
        return self._right.extract_max()
```



Running Time Analysis

- Unbalanced : height $= n = \sum_{i=0}^{n-1} 2^i$
- Balanced : height(left) $\approx$ height(right)
  - → Minimum height $= \log_2(n+1)$
  - ∴ Balanced BST, $\Theta(h) = \Theta(\log n)$

Abstract Syntax Trees.

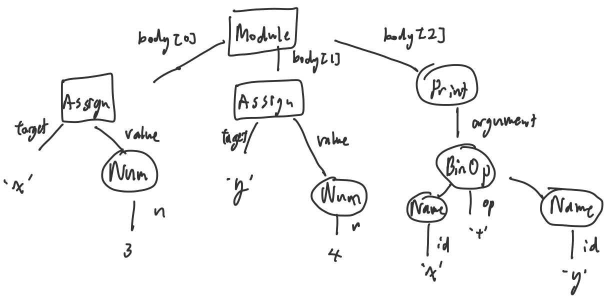
Module (body: list[Statement]) (evaluate)

Statement

- Expr (evaluate(env: dict[str, Any]))
  - Num (n: int | float)
  - BinOp (left: Expr, op: str, right: Expr) (self.left.evaluate(env) +/* self.right.evaluate())
  - Name (id: str) (env[self.id] gives value)
- Assign (target : str → change vaible by env[target], value : Expr)

([self.target] = self.val.evaluate(env))

Abstract Syntax Tree



(env[self.target] = self.value.evaluate(env)

- Print (argument : Expr)
- If (test : Exp.
  body : list [Statement]
  orelse : list [Statement])
- for (Target : str
  Start : Exp.
  Stop : Exp.
  body : list [Statement])

Graphs

$G = (V, E)$

- neighbour : exists an edge between
- degree : number of neighbours
- path : a sequence of distinct vertices
- length : no. of edges of a path
- Connected => exist a path

```
def _Vertex :
    item : Any
    neighbours : set [_Vertex]
    def __init__(self, item: Any, neighbours: set[_Vertex]) -> None:
        self.item = item
        self.neighbours = neighbours
```

```
def check_connected (self, target_item: Any, visited: set[_Vertex]) -> bool:
    if self.item == target.item:
        return True
    else:
        new_visited = visited.union(self)   or, visited.add(self)
        for n in self.neighbours:
            if n not in new_visited:
                if n.check_connected(target_item, new_visited):
                    return True

    return False
```

Properties

- max no. of edges $\leq \frac{|V|(|V|-1)}{2}$
- Transitivity of connectedness
  - If v is connected to both u and w, => u and w are connected.

```
class Graph:

    _vertices: dict [Any, _Vertex]

    def __init__(self) -> None:
        self._vertices = {}
```

A Limit for connectedness.

✲ Trick: Remov

- Max no. of edges $= C^n_2 = \frac{n(n-1)}{2}$
  - Let $n \in \mathbb{Z}^+$. Let $P(n)$ be
    For $\forall G = (V, E)$,
    if $(|V| = n) \wedge \left(|E| \geq \frac{(n-1)(n-2)}{2} + 1\right)$,
    => Then G is connected

Proof.

Base Case: $n = 1$.
$P(1)$ is True vacuously since no $|V| = 1$ and $|E| \geq \frac{1}{2}$, as only one vertex cannot have any edges.

Inductive Step.
Assume $P(k)$ holds. Need to show $P(k+1)$ holds,
i.e. $P(k+1)$: $|V| = k+1$ and $|E| \geq \frac{k(k-1)}{2} +$
=> G is connected.

Case 1:
For $|E| = \frac{(k+1)k}{2}$, G is connected as $|V|$

Case 2:
For $|E| < \frac{(k+1)k}{2}$,
Then there exist at least 2 vertice that is
One of the vertex has at most $k+1$
Remove that vertex,

$$|V|' = |V| - 1 = k + 1 - 1 =$$

$$|E|' = |E| - \text{removed edge}$$

$$\geq |E| - (k-1)$$

$$\geq \frac{k(k-1)}{2} + 1 - k + 1$$

$$= \frac{(k-2)(k-1)}{2} + 1$$

Cycles and Trees

(A graph G either has a cycle, or a tree.)

- Cycle in G : $v_0, v_1, \ldots, v_k$
  - At least 3 vertices. $k \geq 3$
  - $v_0 = v_k$
  - v_i adjacent v_{i+1}.
  - ✲ No cycles : Tree : removing any edge disconnects G.
- Removing an edge from a cycle, still connected.
  - Proof: Let $G' = (V, E - e)$

Trees:

- connected
- no cycles
- $|E| = |V| - 1$

✲ $d(v) = 1$ (If v is at max distance with u)

- assume longest path between v and u.
- → path → at least one neighbor for u
- → end → v can only be adjacent to the one before, or else form cycle or extend path, contradicts.

Spanning Trees.

- A $\underbrace{\text{spanning tree}}_{\text{is a tree}}$ of a larger graph

Brute force:

```
    edges_so_far = all edges in self
    while edges_so_far contains a cycle:
        remove an edge in the cycle from edge_so_far

    return edges_so_far
```

Spanning Tree Algorithm.

```
    def spanning_tree(self, visited : set[_Vertex]) -> list[set]:

        edges_so_far = []
        visited.add(self)
            if n not in visited:
                edges_so_far.append({self.item, n.item})
                edges_so_far.extend(n.spanning_tree(visited))

        return edges_so_far
```

Sorting

Selection Sort

- Finds the smallest in the unsorted list
- Put it at front.

```
def selection_sort (lst : list) → None:   Θ(n²)
    for i in range (0, len(lst)) :
        index_of_smallest = _min_index (lst, i)
        lst [index_of_smallest], lst[i] = lst[i], lst[index_of_smallest]

def _min_index (lst, i) → int:   Θ(n-i)
    "Return smallest's index in lst[i:]"
    index_of_smallest = i                                  ] 1
    for j in range (i+1, len(lst)):                        ] n-i-1
        if lst [j] < lst [index_of_smallest]:              ] 1
            index_of_smallest = j
    return index_of_smallest.                              ] 1
```

5 i=2 8 3

Θ(n²) regardless, find smallest Θ(n-i), with Θ(n) times

Running-time Analysis

- Let input list size = n.
- For _min_index :
  - ① constant step = 1.
  - ② Loop iterates n-i-1 times. each loop has 1 step.

  ∴ Total step $= (n-i-1)\cdot 1 + 1$
  $= n-i$
  $\in \Theta(n-i)$

- For selection_sort :
  - ① Loop runs n times
    - → calls _min_index = n-i
    - → swapping takes 1 step

    $\}$ n-i+1 (1, 2, ..., n)

$$\therefore \text{Running-time} = \sum_{i=0}^{n-1} n-i+1 = n(n+1) - \sum_{i=0}^{n-1} i$$
$$= n(n+1) - \frac{n(n-1)}{2}$$
$$= n^2+n - \frac{n^2}{2} + \frac{n}{2}$$
$$= \frac{1}{2}n^2 + \frac{3}{2}n$$
$$\in \Theta(n^2)$$

Insertion Sort

- Insert the element into correct spot in the sorted list.

```
def insertion_sort (lst: list) → None:
    for i in range (0, len(lst)):        Θ(n²) worst
        _insert( lst, i)                 Θ(n) if sorted

def _insert (lst, i):
    j = i
    while not (j == 0 or lst [j-1] <= lst [j]):
        lst [j-1], lst [j] = lst[j], lst[j-1]
        j = j-1
```

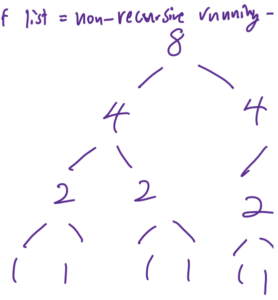
Merge Sort Always: Θ(nlogn)

- Divide recursively (Easy!)
- Combine sorted half together (Hard!)

```
def mergesort (lst : list): → list:
    if len(lst) < 2 :
        return lst. copy ()

    else :
        mid = len(lst) // 2
        left = mergesort ((lst [ : mid ]))       n/2 steps
        right = mergesort ((lst [ mid: ]))       - n/2 steps } n steps
        return _merge (left, right)
```

```
def _merge (lst 1, lst 2 ) → list:
    i1, i2 = 0, 0
    sorted_so_far = []
    while i1 < len(lst1) and i2 < len(lst2) :
        if lst[i1] ≤ lst [ i2] :
            sorted_so_far.append ( lst[i1])
            i1 += 1
        else:
            sorted_so_far.append ( lst [i2])
            i2 += 1
    if i1 == len(lst) :
        return sorted_so_far + lst2[i2 : ]   # non-mutating
    else :
        return sorted_so_far + lst1 [i1: ]
```

n = length of list = non-recursive running -



Each level : n steps

No. of levels : $\log_2 n$ +

Total work

$= n \times (\log_2 n + 1)$

$\in \Theta(n \log n)$

pivot is extreme

pivot is median

pivot is

$\in \Theta(n\log_2 n)$

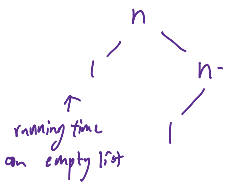
Quicksort Best : $\Theta(n\log n)$ Worst : $\Theta(n^2)$

- pick pivot : partition into smaller/greater lists. (Hard!)
  put pivot in middle.
  → recursively combine (Easy!)

```
def quicksort(lst: list) → list:
    if len(lst) < 2:
        return lst.copy()
    else:
        pivot = lst[0]
        small, big = _partition(lst[1:], pivot)
        return quicksort(small) + [pivot] + quicksort(big)
```

```
def _partition(lst, pivot) → tuple[list, list]:
    smaller = []
    bigger = []

    for item in lst:
        if item <= pivot:
            smaller.append(item)
        else:
            bigger.append(item)
    return smaller, bigger
```

Extreme pivot: n is



Worst: $(1+2+\dots+n)+$

Inplace - partition

- use small_i to compare To pivot
- If lst[small_i] <= pivot, small_i add 1.
- If lst[small_i] > pivot, swap with lst[big_i] and (big_i - 1)

```
def _in_place_partition(lst: list) → None:
    pivot = lst[0]
    small_i = 1
    big_i = len(lst)    # Lager than biggest index by 1.
    while small_i != big_i:
        if lst[small_i] <= pivot:
            small_i += 1                                              # Evaluate the next
        else:
            lst[small_i], lst[big_i - 1] = lst[big_i - 1], lst[small_i]   # Swap current with big_i spot
    lst[0], lst[small_i] = lst[small_i], lst[0]                   # Swap pivot with the last entry in small
    return small_i                                                # position of pivot.
```