Summary of Data Structures

Jake Underland (1A193008-2)

1/30/2022

目 次

概 要

本レポートでは kd 木，フィボナッチヒープ，AVL 木，接尾辞配列の 4 つのデータ構造について解説していく．

1 k-d 木

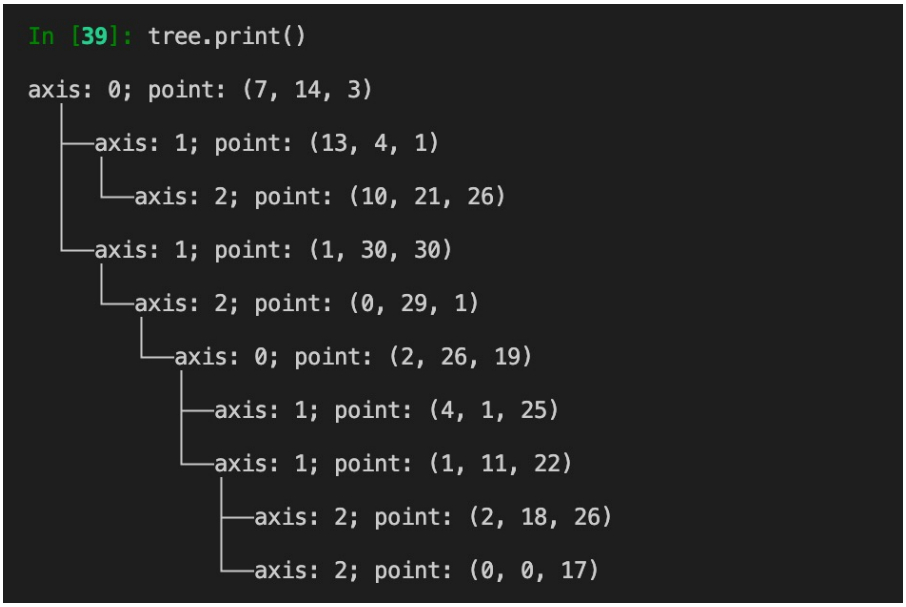
k-d 木とは，二分木の一つであり，全てのノードが k 次元のユークリッド空間上の点であるものをいう．二分木であるため，全てのノードには 2 つの子供ノード (左，右）がある．k 次元の座標空間のいずれかの軸 (例:2 次元空間の x 軸) に着目し，根ノードの軸の値と新しく挿入するノードの軸の値を比較して挿入するノードの方が低ければ，根ノードの左部分木に挿入し，高ければ，根ノードの右部分木に挿入する．これを，親から辿っていき全てのノードを部分木とみなして再帰的に行うことで挿入することができる．一般的には，深さが増すごとに基準となる軸を交互に変えていく．これが意味することは，すなわち全ての葉ではないノードは，座標空間を 2 つの半空間に分割する超平面を作り出しており，そのノードの下に属するノードは 2 つに仕分けられる．ここまでのロジックを組んだコードを python で実装してみたのが以下だ．

```python
class kdtree(object):
    def __init__(self, point, axis=0):
        self.point = point # k次元の座標 (tuple)
        self.k = len(self.point)
        self.axis = axis % self.k  # 比較する軸を increment することで交互に入れ替える
        self.left_child = None
        self.right_child = None
        self.children = []

    def insert(self, point):
        assert len(point) == self.k
        if point[self.axis] > self.point[self.axis]:
            if self.right_child:
                self.right_child.insert(point)
            else:
```

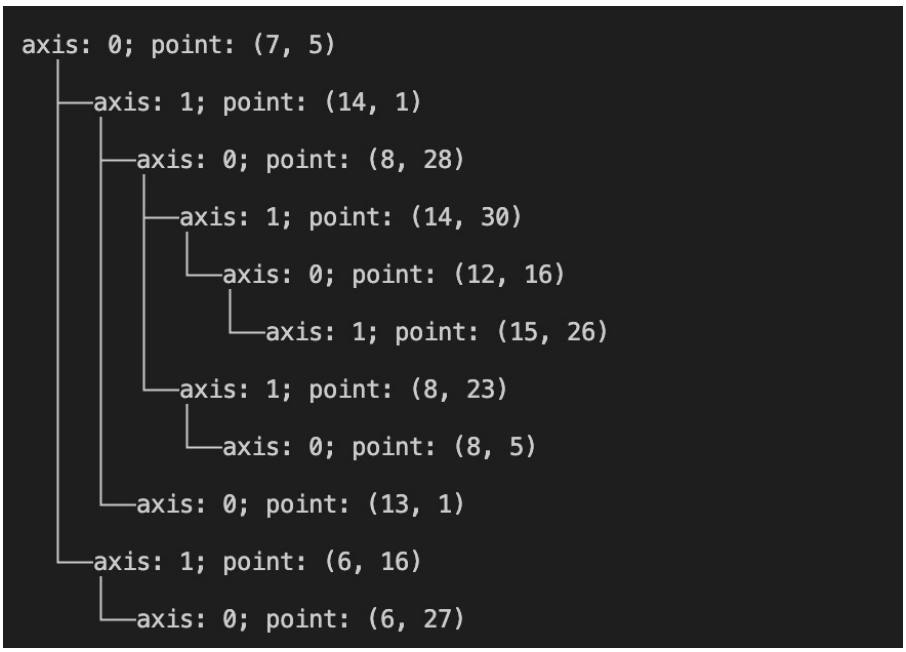
```
                self.right_child = kdtree(point, self.axis + 1)
                self.children.insert(0, self.right_child)
        else:
            if self.left_child:
                self.left_child.insert(point)
            else:
                self.left_child = kdtree(point, self.axis + 1)
                self.children.append(self.left_child)
```

これを可視化したのが以下になる．ただし，ターミナルでの可視化の都合上，右の子ノードを上，左の子ノードが下に来るように，木全体を左方向に 90 度傾けた様子となっている．

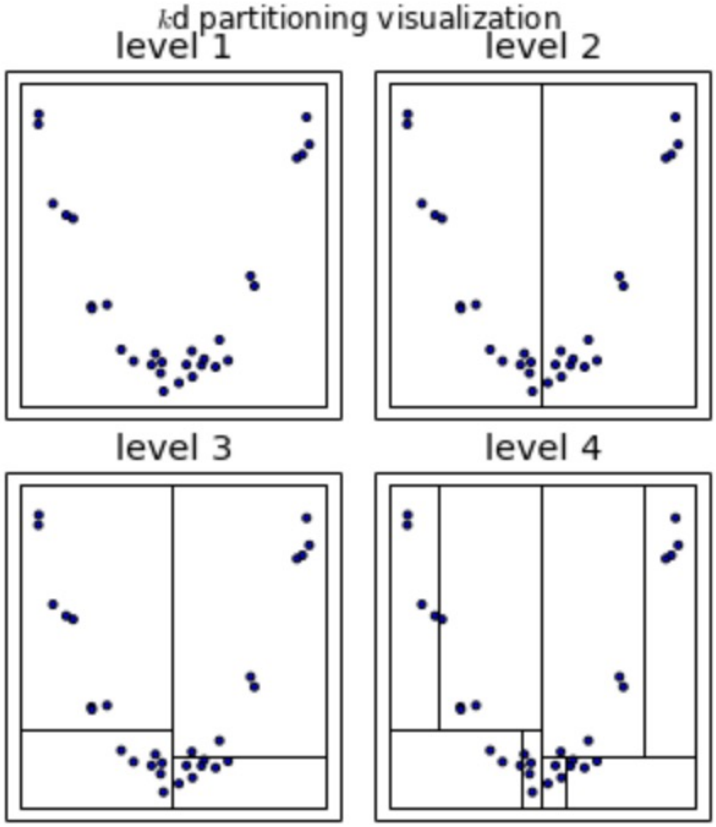


axis が参照される座標軸を指し，深度が増すごとに入れ替わっているのが観察できる．また，全てのノードにおいて，その下に属するノードは，指定された座標軸で 2 つの半空間に分割されていることが見て取れる．

kd 木は多次元データを扱う際に有用である．例えば，簡略化のため，下の 2 次元の kd 木に注目する．



(14，23) に最も近い（ユークリッド距離が最小な）点を探したいとする．この時, 根っこから kd 木を辿ってみる．$14 > 7$ なので，右の（上の）部分木に移る．$23 > 1$ なので，また右の部分木に移る．この移動を繰り返すたびに，探索する範囲は狭くなっていき，最終的に直近点の近傍に落ち着き，その範囲内の点を探すことにより直近点を割り出すことができる．このような繰り返した領域のパーテーションが kd 木の特徴である．したの図は，2 次元空間におけるこのパーテーションの反復を可視化したものだ．



2 フィボナッチヒープ

フィボナッチヒープとは，マージ可能なヒープ構造をしており，以下の操作を行うことができる:

- ヒープ作成：ヒープを作る
- 挿入：ヒープに追加する
- 最小値：ヒープの中の最小値へのポインターをリターンする
- 最小値抽出：ヒープの中の最小値へのポインターをリターンし，最小値をヒープから削除する
- マージ：二つのヒープをマージする
- 要素の下方更新：ヒープのある要素の値を，それ以下の値に更新する
- 削除：ヒープから要素を削除する．

フィボナッチヒープの利点としては，これらの操作のうちの多くを定数時間で行うことができることだ．二分ヒープと比較すると以下のようになる．

	二分ヒープ	フィボナッチヒープ
ヒープ作成	$O(1)$	$O(1)$
挿入	$O(\log n)$	$O(1)$
最小値	$O(1)$	$O(1)$
最小値抽出	$O(\log n)$	$O(\log n)$
マージ	$O(n)$	$O(1)$
要素の下方更新	$O(\log n)$	$O(1)$
削除	$O(\log n)$	$O(\log n)$

フィボナッチヒープの構造は，最小ヒープの性質（子ノードの方が親ノードより値が大きい）を持った木の集合である．二項ヒープと違い，この集合にの木の形に規定はなく，また同じ形をした木が複数あっても構わない．そのため，木のマージや挿入は，単純に木をつなぎ合わせれば完了する．これがこれらの操作の高速化の理由である．また，要素の下方更新はその要素を独立の木として切り離して行われることもあるため，この操作も高速化される．

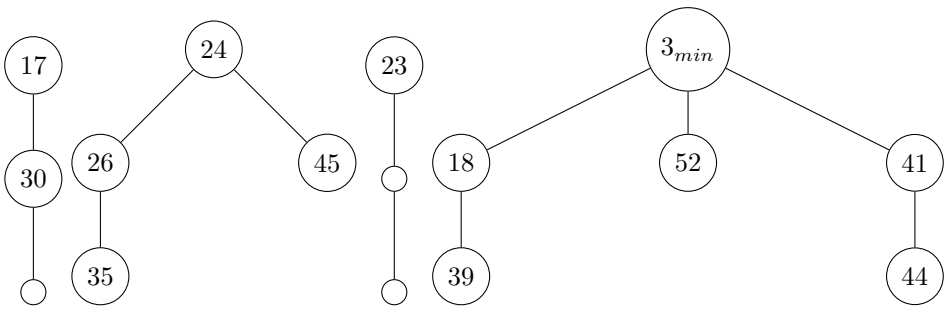


図 1: フィボナッチヒープ（但し空のノードは高さを揃えるためのもの）.

フィボナッチヒープは最小値を示すポインターの情報を保存しており，ヒープ内の全ての木の最小値が連結リスト（双方向循環リスト）によりつながっているため，一つのポインターで全ての木の最小値をアクセスすることができる．図 2 では描かれていないが，根のノードは全て連結されている．

フィボナッチヒープで最も複雑な操作は最小値の抽出と削除である．最小値の抽出は，以下のように進む：

1. 最小ノードを削除する

2. 削除したノードの部分木の根を全て根の連結リストに加え，最小のものを先頭に持ってくる．
3. 根のリスト上で次元が等しい木をマージする (二項ヒープと同様)．これを，全ての木の次元が異なるまで続ける．
4. 残った根の中から最小値を見つけ，ポインターを指す.

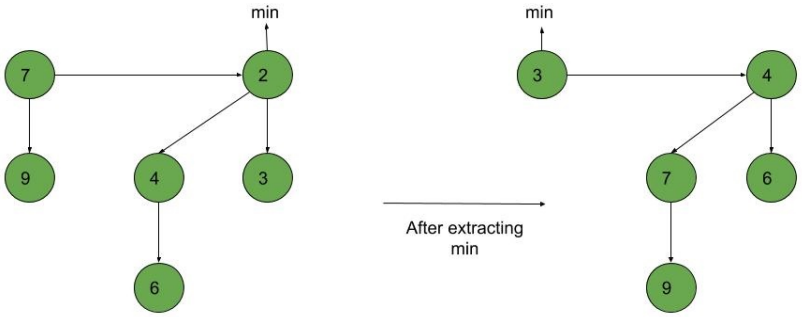


図 2: 最小値抽出

削除操作は削除する要素の値を $-\infty$ に更新してから上の操作を行えばいい．

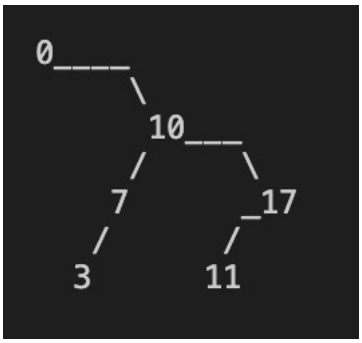
フィボナッチヒープはヒープのマージ，挿入，下方更新を多く行う必要があるアルゴリズムによく使われる．

3　AVL 木

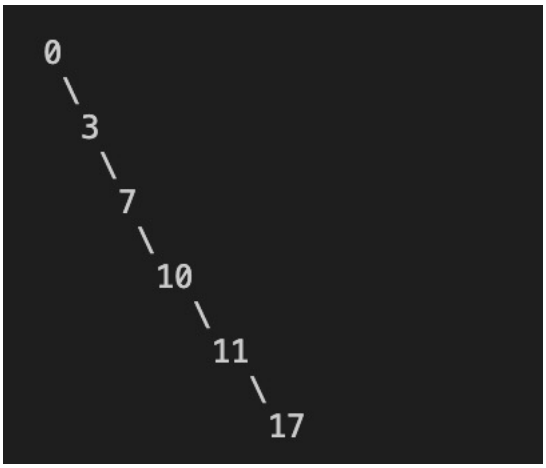
AVL 木とは二分探索木の一種で，どのノードの左右部分木の高さも差が 1 以内のものを指す．このような条件を満たす木を平衡二分探索木というが，AVL には，何らかの操作によってどこかで左右部分木の高さに 1 を超える差が生じても，再び平衡となるように自身を再構成できることが特徴である．

通常の 2 分探索木は，それぞれの頂点が最高 2 つの子を持つ二分木で，次の特徴を満たす:
全ての頂点 v において，v の左部分木に含まれるどの頂点 `v_l` に対しても `v.key() >= v_l.key()` であり，v の右部分木に含まれるどの頂点 `v_r` にたいしても `v.key() <= v_r.key()` が成り立つ．

この性質の結果，二分探索木の構造は要素を挿入する順序に大きく依存する（図 3 参照）．



(a) 一般的な二分探索木

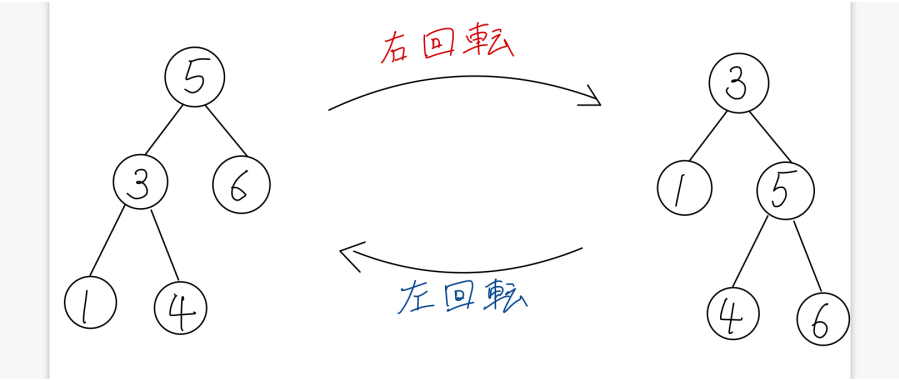


(b) ソートされた順に挿入した時の二分探索木

図 3: 同じ整数の集合でも，挿入の順序で木の形が大きく変わる．

バランスが比較的保たれた二分探索木では，$2^h = N$ であり，なお二分探索木への操作のほとんどは $O(h)$ なので，結果 $O(\log N)$ となるしかし，図 3 の (b) の場合は，$h = N$ となっているため，操作の計算量は $O(N)$ と大きく増加する．

この問題を，AVL 木は解消する．AVL 木は，削除，挿入といった操作の後に，必ず右部分木の高さと左部分木の高さの差を確認する．差が 1 を超えるようであれば，木は回転を行う．木の回転には右回転と左回転があり，それぞれは要素の順番を崩さずに右回転であれば右部分木の高さを 1 増やして左部分木の高さを 1 減らし，左回転であれば左部分木の高さを 1 増やして右部分木の高さを 1 減らす．その様子は，下の図に表される



AVL 木は回転を通して常に平衡であるため，確実に $O(\log N)$ に近い計算量で操作を実現している．

4　接尾辞配列

接尾辞配列とは，ある文字列の接尾辞を全て抽出し，それをアルファベット順に並べた時の，各接尾辞の位置を保存した配列である．以下のコードは，python でこれを実装したものである．

```
def extract_suffix_array(string):
    n = len(string)
    suffixes = []
    for i in range(n):
        suffixes.append((i, string[i:]))
    sorted_suffixes = sorted(suffixes, key=lambda x:x[1])
    for i in range(n):
        print suffixes[i], " " * i, "   --->  ", sorted_suffixes[i]
    return [x[0] for x in sorted_suffixes]

string = "algorithms"
suffix_array = extract_suffix_array(string)
```

```
## (0, 'algorithms')     --->   (0, 'algorithms')
## (1, 'lgorithms')      --->   (2, 'gorithms')
## (2, 'gorithms')       --->   (7, 'hms')
## (3, 'orithms')        --->   (5, 'ithms')
## (4, 'rithms')         --->   (1, 'lgorithms')
## (5, 'ithms')          --->   (8, 'ms')
## (6, 'thms')           --->   (3, 'orithms')
## (7, 'hms')            --->   (4, 'rithms')
## (8, 'ms')             --->   (9, 's')
## (9, 's')              --->   (6, 'thms')
```

```
print("suffix array of '{}' is: {}".format(string, suffix_array))
```

```
## suffix array of 'algorithms' is: [0, 2, 7, 5, 1, 8, 3, 4, 9, 6]
```

接尾辞配列の代表的な応用例としては，文字列の中の部分文字列の出現回数の探索があげられる．接尾辞配列は全ての可能な接尾辞を含んでいるので，この文字列の中で考えられる部分文字列は，必ずいずれかの接尾辞の接頭辞として存在しているはずだ．また，接尾辞配列ではアルファベット順に並んでいるので，該当するような接尾辞は隣り合わせになっているはずである．よって，二分探索法を 2 回行うこと（接尾辞配列で最初に部分文字列を含む接尾辞と最後に部分文字列を含む接尾辞）で，効率よく目的の部分文字列の出現回数を判断することができる．

5　References

Cormen, T.H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to algorithms (3rd ed.). MIT Press.

VanderPlas, J., Ivezic, Z., Connolly, A., Gray, A. (2013). “Statistics, Data Mining, and Machine Learning in Astronomy”. Princeton University Press.

https://qiita.com/flare/items/20439a1db54b367eea70

https://www.tutorialspoint.com/data_structures_algorithms/avl_tree_algorithm.htm