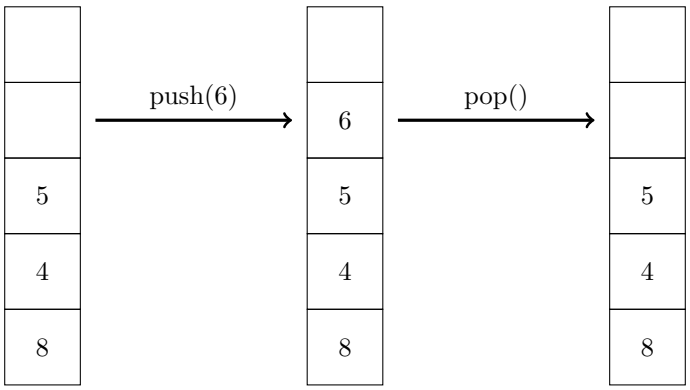
EEIC2024 アルゴリズムの授業で扱ったデータ構造のアルゴリズムのシケプリです。扱った内容は以下の通りです。

- スタック
- キュー
- 線形リスト
- ツリー
- ヒープ
- セグメント木 (発展的な内容)
- BIT(発展的な内容)

# I. スタック

スタックは LIFO(Last In First Out) といって、最後に入れたデータが最初に取り出されるデータ構造です。本を積み重ねるイメージで考えるとわかりやすいです。スタックには以下の操作があります。

- push: スタックにデータを追加する
- pop: スタックからデータを取り出す




**スタックのイメージ**

## i. スタックの実装

実装のポイントは以下の 3 つです。Python の class を使って実装します。

- スタックは配列で実装する
- 要素が入る位置を示すポインタを持つ

- push と pop のメソッドを実装する

実装は以下の 2 のようになります。

**コード 1　スタックの実装**

```
  class Stack:
    def __init__(self, size: int) -> None:
        self.stack: list[int] = [0] * size
        self.top: int = 0

    def push(self, value: int) -> None:
        if self.top < len(self.stack):
            self.stack[self.top] = value
            self.top += 1
        else:
            print("This stack is full.")

    def pop(self) -> None:
        if self.top > 0:
            self.top -= 1
            return_value = self.stack[self.top]
            return return_value
        else:
            print("This stack is empty.")

def main():
    size = 5
    stack = Stack(size)

    stack.push(8)
    stack.push(4)
    stack.push(5)
    stack.push(6)
    return_value = stack.pop()

    print(f"returned value is {return_value}")

main()
```

Column1 collections.deque

実際にスタックを使用するときは、Python の標準ライブラリである collections の deque を使うと便利です。Python では Stack スタックやキューが明示的に用意されていないため、双方向キューである deque を使ってスタックやキューを使用します。スタックは上記のように配列で簡単に実装可能ですが、標準ライブラリの deque を使うことで、より高速にスタックを使用できます。

**コード 2　スタックの実装**

```
    def main():
    from collections import deque
    stack = deque()

    stack.append(8)
    stack.append(4)
    stack.append(5)
    stack.append(6)

    return_value = stack.pop()

    print(f"returned value is {return_value}")

main()
```

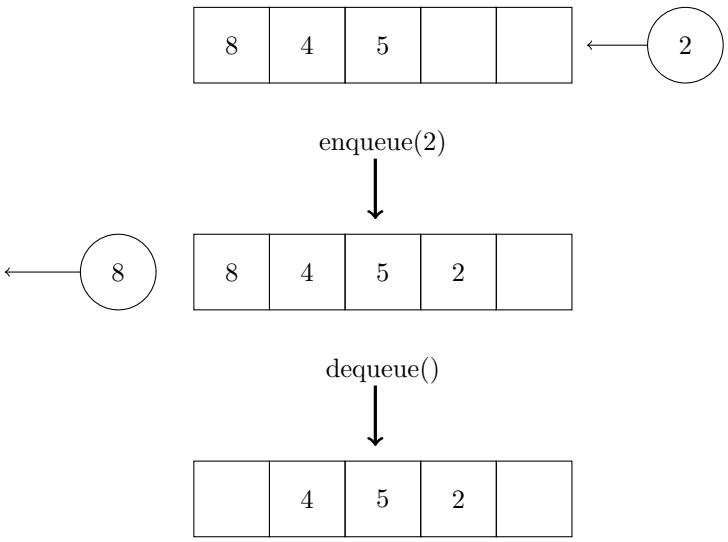
## II.　キュー

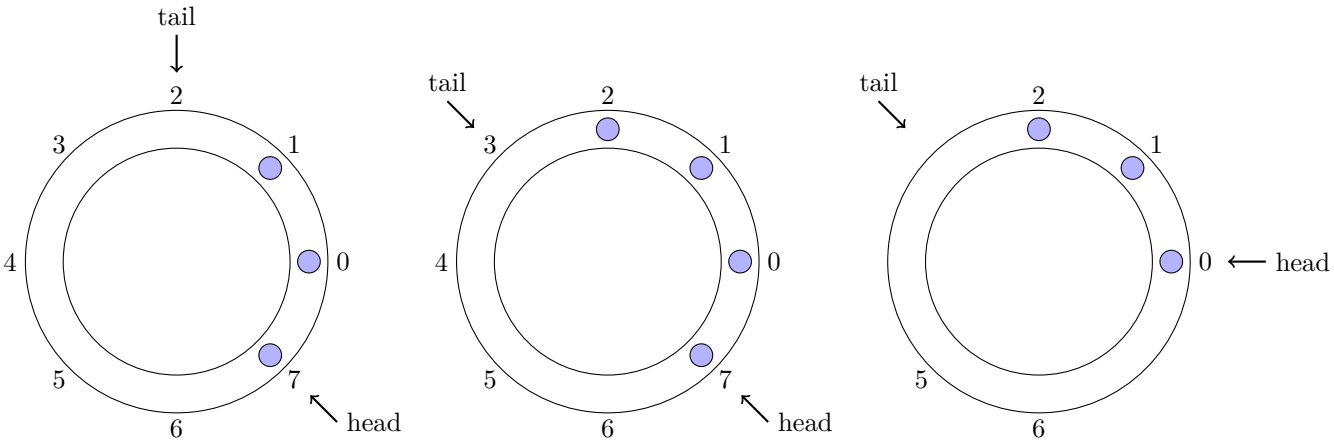
キューは FIFO(First In First Out) といって、最初に入れたデータが最初に取り出されるデータ構造です。ディズニーランドの待ち行列のようなイメージで考えるとわかりやすいです。キューには以下のような操作があります。

- enqueue: キューにデータを追加する
- dequeue: キューからデータを取り出す

先ほどのスタックと違って配列の頭からデータを取り出していることに注意してください。つまり、dequeue をするたびに取り出すデータの位置をずらす必要があります。従って、データを取り出すための先頭アドレス、データを追加するための末尾アドレスを持つ必要があります。

しかし、ここで困ったことがあります。dequeue をするたびに配列の頭が右にずれていくと、配列の先頭にデータが残っているにも関わらず、配列の末尾にデータが追加できなくなります。この問題を解決するために、リングバッファというデータ構造を使います。






**enqueue()**　　**dequeue()**

リングバッファの仕組みを見てみましょう。enqueue をするときは tail の位置にデータを追加し tail を 1 だけ進めます。dequeue をするときは head の位置のデータを取り出し head を 1 だけ進めます。head と tail が同じ位置になったときは、キューが空になります。実装の際は、要素の数で割った余りを使って head と tail の位置を管理します。

### i.　キューの実装

キューの実装のポイントは以下の 3 つです。

- キューは配列で実装する
- 取り出す位置を示すポインタと追加する位置を示すポインタの 2 つを持つ
- リングバッファをモジュロ演算で実装する (%を使う)
- enqueue と dequeue のメソッドを実装する

**コード 3　キューの実装**

```
1   class Queue:
2     def __init__(self, size: int) -> None:
3         self.queue = [0] * size
4         self.size = 0
5         self.head = 0
6         self.tail = 0
7
8     def enqueue(self, value: int) -> None:
9         if not self.is_full():
10            self.size += 1
11            self.queue[self.tail] = value
```

```
12              self.tail = (self.tail + 1) % len(self.queue)
13          else:
14              print("queue is full")
15
16      def dequeue(self) -> int:
17          if not self.is_empty():
18              self.size -= 1
19              return_value = self.queue[self.head]
20              self.head = (self.head + 1) % len(self.queue)
21              print(return_value)
22          else:
23              print("queue is empty")
24
25
26      def is_empty(self):
27          return self.size == 0
28
29      def is_full(self):
30          return self.size == len(self.queue)
31
32  queue = Queue(5)
33
34  queue.enqueue(8)
35  queue.enqueue(4)
36  queue.enqueue(5)
37
38  queue.enqueue(2)
39
40  queue.dequeue()
```

**Column2 collections.deque**

スタックと同様にキューも実際に使用するときは、Python の標準ライブラリである collections の deque を使うと便利です。

```
1       from collections import deque
```

```
2
3  queue = deque()
4
5  queue.append(8)
6  queue.append(4)
7  queue.append(5)
8
9  queue.append(2)
10
11 return_value = queue.popleft()
12
13 print(f"returned value is {return_value}")
```

## III.　線形リスト

線形リストとは、以下の図のようにデータと次のポインタを持つデータを連結させるデータ構造です。線形リストは必要なときに必要なデータを追加していくことができるため、空間計算量的に効率がいいです。しかし、データを取り出すときは先頭から順番にたどる必要があるため、時間計算量的には効率が悪いです。

今回実装する線形リストは以下の操作ができるものです。

- 片方向の線形リスト
- 新しい要素は常に末尾に追加する
- 検索は先頭から順番に行う

双方向の線形リストや、データの追加や削除を任意の位置で行うものもありますが、今回は片方向の線形リストを実装します。


### i.　線形リストの実装

先ほどのスタックやキューは配列を使って実装しましたが、線形リストはポインタ (Python ではクラスのインスタンス) を使って要素をつなげて実装します。

**コード 4　線形リストの実装**

```
class Node:
    def __init__(self, value) -> None:
        self.value = value
        self.next: Node = None

class LinkedList:
    def __init__(self) -> None:
        self.head = Node(None)

    def append(self, value: int) -> None:
        current_cell = self.head
```

```
12          while current_cell.next != None:
13              current_cell = current_cell.next
14
15          # 追加する数値を値にもつノード
16          new_cell = Node(value)
17          # 現在の最後のノードの次のノードに付け加える
18          current_cell.next = new_cell
19
20      def pop(self, value: int) -> Node | None:
21          """
22          valueと同じ数値を値に持つノードを取り除く
23          """
24          prev_cell = None
25          current_cell = self.head
26
27          while current_cell != None:
28              if current_cell.value == value:
29                  # 取り出すため、付け替える
30                  prev_cell.next = current_cell.next
31                  return value
32              else:
33                  prev_cell = current_cell
34                  current_cell = current_cell.next
35
36          return None
37
38  linked_list = LinkedList()
39
40  linked_list.append(10)
41  linked_list.append(535)
42  linked_list.append(51)
43  linked_list.append(40)
44  linked_list.append(40)
45  linked_list.append(1)
46
47  return_value = linked_list.pop(51)
```

```
48
49 print(return_value)
```

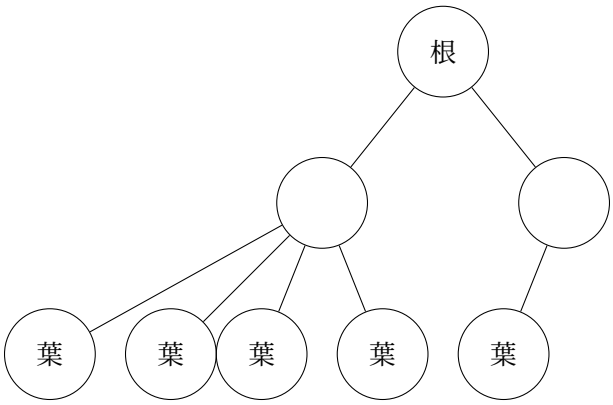
# IV.　ツリー (木構造)

木をひっくり返して根っこを上にして書いた木のような離散構造をツリーと言います。ツリーは用語が重要なため、以下の用語を覚えておくと良いです。

### i.　木構造の用語

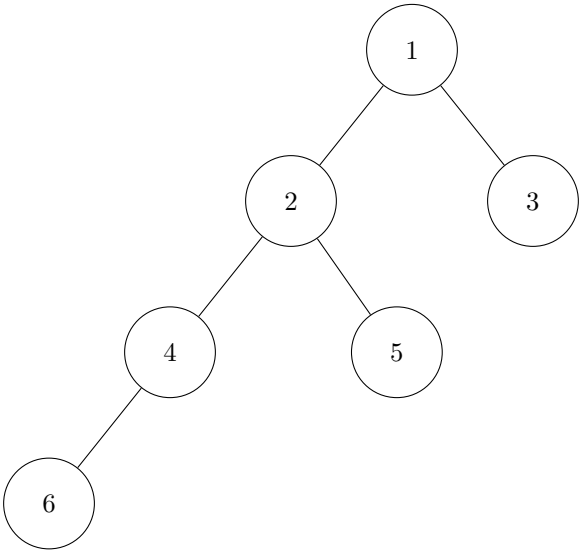
#### 1.　根と葉

一番上のノードを**根**、一番下のノード (より下に子ノードを持っていないノード) **葉**といいます。




#### 2.　深さと高さ

あるノードから繋がっている葉までの自分を除いたノードの数のうち最大の個数を**高さ**といいます。 葉ノード自体の高さは 0 です。また、根からあるノードまでの高さを**深さ**といいます。



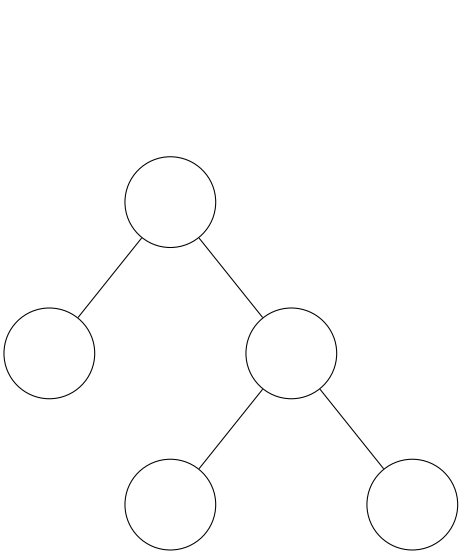
### 3.　二分木 (binary tree)

すべてのノードに関して、その子ノードの数が 2 以下である木を**二分木**といいます。二分木は以下のような特徴があります。上の木も二分木になっています。

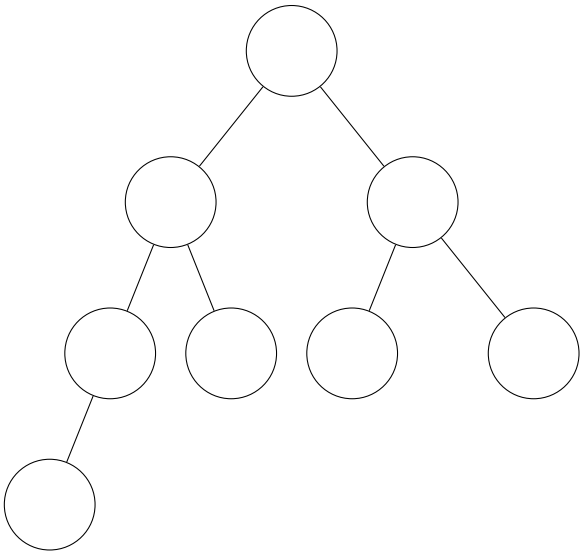
また、、完全二分木というものがあります。完全二分技は英語で表記すると full binary tree と complete binary tree があります。これらは独立した定義ですので、注意してください。違いを整理すると以下のようになります。

- full binary tree: すべてのノードが 0 個または 2 個の子ノードを持つ二分木
- complete binary tree: 根がある階層 (level) を除いて、すべての階層で埋まりうるノードがすべて埋まっているかつ、最後の階層のノードは左から右に向かって埋まっている二分木

両者の違いは最初は分かりにくいです、以下の図を見て理解を深めてください。



full binary tree の例



complete binary tree の例

## V.　ヒープ

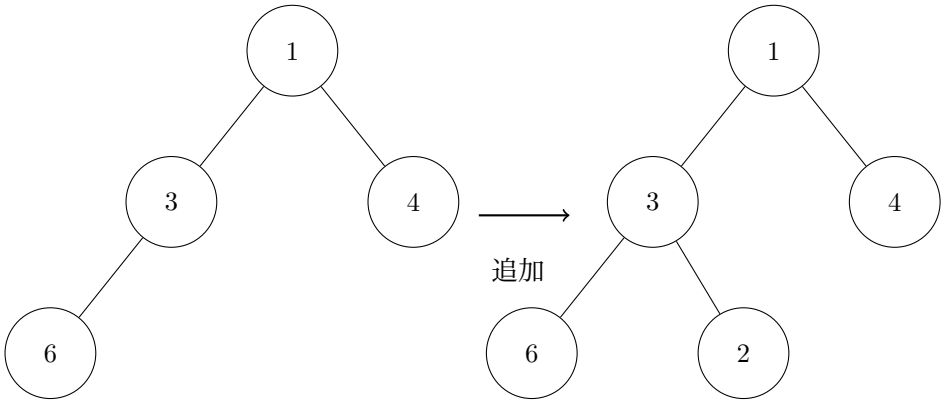
木構造を使ったデータ構造にヒープがあります。ヒープは以下の特徴があります。

- 親ノードは子ノードよりも大きい (または小さい) という性質を持つ
- 根ノードは常に最大 (または最小) となる

さらに今回はヒープの中でさらに、complete binary tree になっている**二分ヒープ**というデータ構造を扱います。ヒープは complete binary tree に対して、以下の操作を行います。

- 要素の追加
- 要素の取り出し

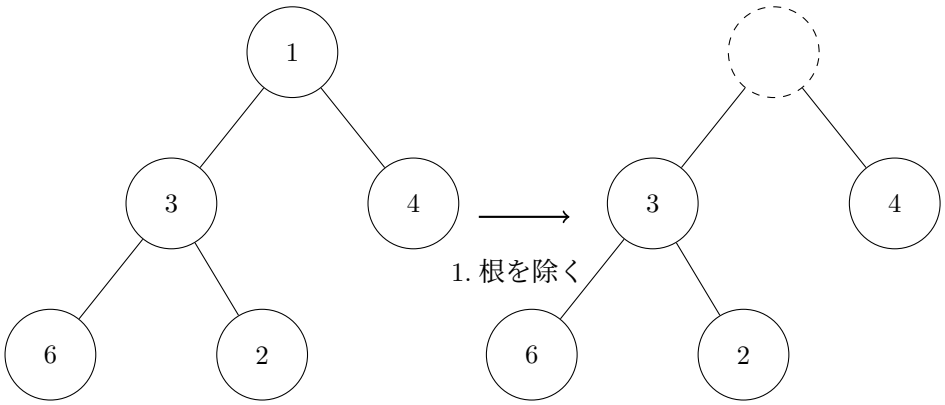
要素の追加に関して、二分ヒープが complete binary tree になっているため、根がある階層 (level) の一番右に要素を追加してきます。追加時に、追加したノードから根までヒープの条件を満たすように再帰的にノードを交換していきます。以下は最小ヒープの例です。

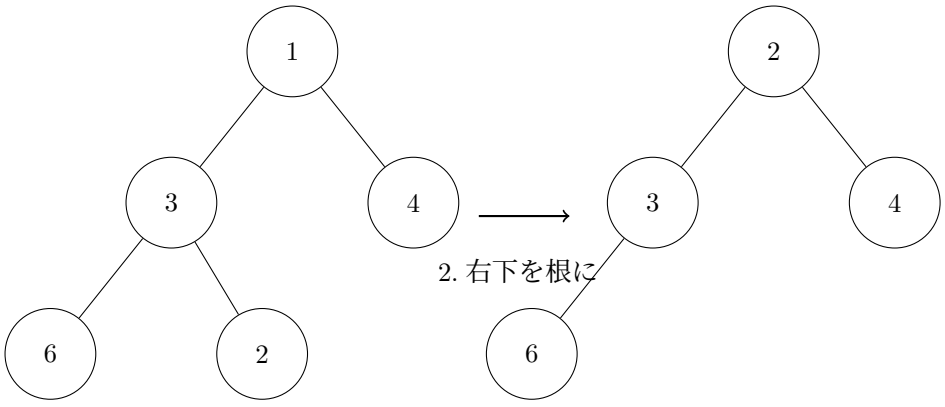



二分ヒープからの削除は少し複雑です。以下のような操作を行います。

1. 根ノードを取り除く
2. 一番右下のノードを根に移動する
3. 根から下に向かって、子ノードと比較して、ヒープの条件を満たすように再帰的にノードを交換していく

一番右のノードを根に移動する理由は、complete binary tree を保つためです。もしも、根を取り除いた後に直接根の子ノードを根にしてしまうと、必ずしも complete binary tree にならず実装が複雑になります。


1
3
4
6
2
1. 根を除く
3
4
6
2


1
3
4
6
2
2. 右下を根に
2
3
4
6

上の例では、根を取り除き一番右のノードを根に移動すると木がヒープの性質を満たしていますが、一般的にはヒープの性質を満たすまで根から下に向かってノードを再帰的に交換していきます。

### i.　ヒープの実装

二分ヒープの実装のポイントは

- 1-indexed の配列で実装する (要素数 + 1 の大きさの配列を用意する)

二分木は構造体を使わず配列だけで実装できる点が利点です。以下は最小ヒープの実装です。

**コード 5　二分ヒープの実装**

```
class Heap:
  def __init__(self, size: int) -> None:
    self.inf = 1 << 60
    self.size = size + 1
    self.heap = [self.inf] * self.size
    self.tail = 0 # 現在のヒープのデータ数

  # 複数回swap処理が登場するためメソッドにした
  def swap(self, index1: int, index2: int) -> None:
    self.heap[index1], self.heap[index2] = self.heap[index2], self.heap[
        index1]

  def push(self, value: int) -> None:
    if self.tail != self.size - 1:
      self.tail += 1
      self.heap[self.tail] = value
      self.check_from_leaf(self.tail)

  def check_from_leaf(self, node: int) -> None:
    # nodeが根なら再帰終了
    if node == 1:
      return

    parent = node // 2
    # ヒープの条件を満たさない
    if self.heap[parent] > self.heap[node]:
```

```
26        self.swap(parent, node)
27        self.check_from_leaf(parent)
28
29
30    def pop(self) -> int | None:
31      if self.tail != 0:
32        popping_value = self.heap[1]
33        # 一番右の葉ノードを根ノードに移動
34        self.heap[1] = self.heap[self.tail]
35        self.tail -= 1
36        self.check_from_root(1)
37        return popping_value
38      else:
39        return None
40
41    def check_from_root(self, node: int) -> None:
42      left_child, right_child = 2 * node, 2 * node + 1
43
44      # left > rightよりこれだけで十分
45      if left_child > self.tail:
46        return
47
48      # 2つの子ノードの大小で区別する
49      smaller_node = larger_node = 0
50      if self.heap[left_child] < self.heap[right_child]:
51        smaller_node, larger_node = left_child, right_child
52      else:
53        smaller_node, larger_node = right_child, left_child
54
55      # 親ノードと小さい方の子ノードを比較
56      if self.heap[node] > self.heap[smaller_node]:
57        self.swap(node, smaller_node)
58        # 小さい方だけで十分
59        self.check_from_root(smaller_node)
```

# VI.　セグメント木

セグメント木はある条件を満たす演算に関する区間の計算結果を高速に求めるデータ構造です。区間の計算結果を高速にもとめると聞いて累積和や尺取法を思い浮かべるかもしれませんが、セグメント木は扱うデータの要素が逐一変わるような状況でも区間の計算結果を高速に求めることができます。具体的な演算は、区間の最大・最小値や、合計値などがあります。

### i.　数学の知識の確認

セグメント木の内容に入る前に、セグメント木を理解するのに必要な数学の知識を整理します。

#### 1.　二項演算

自然数や文字列の集合 $S$ の任意 2 つの元から新たに値を生成する操作のことを $S$ **上の二項演算**といいます。特に、$x, y \in S$ に対する二項演算を $op(x, y)$ と書き、

$$op(x, y) = x * y$$

と表現することにします。

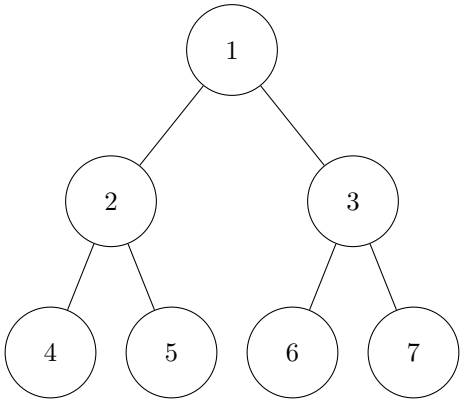
#### 2.　閉じた演算

$x, y \in S$ に対して $op(x, y) \in S$ を満たすような演算を閉じた演算といいます。例えば、自然数の集合 $S$ において、$op(x, y) = x + y$ は閉じた演算ですが、$op(x, y) = x - y$ は閉じた演算ではありません。

今回実装するセグメント木は、閉じた演算を扱うため、セグメント木の演算は閉じた演算になるようにします。

#### 3.　完全二分木

セグメント木は完全二分木になっています。以下の図のように根以外のすべてのノードが 2 つの子ノードを持っていて、すべての葉の深さが同じ木になっています。



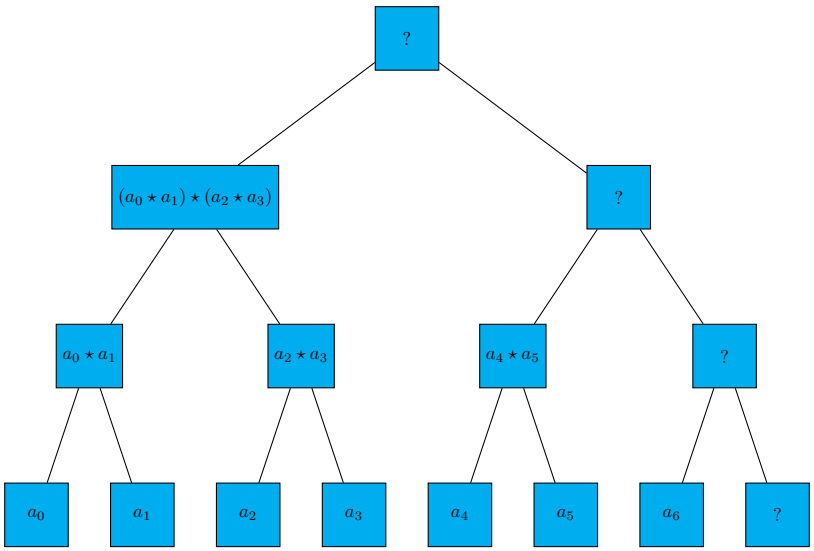
### ii.　セグメント木の仕組み

セグメント木の理解に必要な知識の整理が終わったので、セグメント木の内容に入ります。

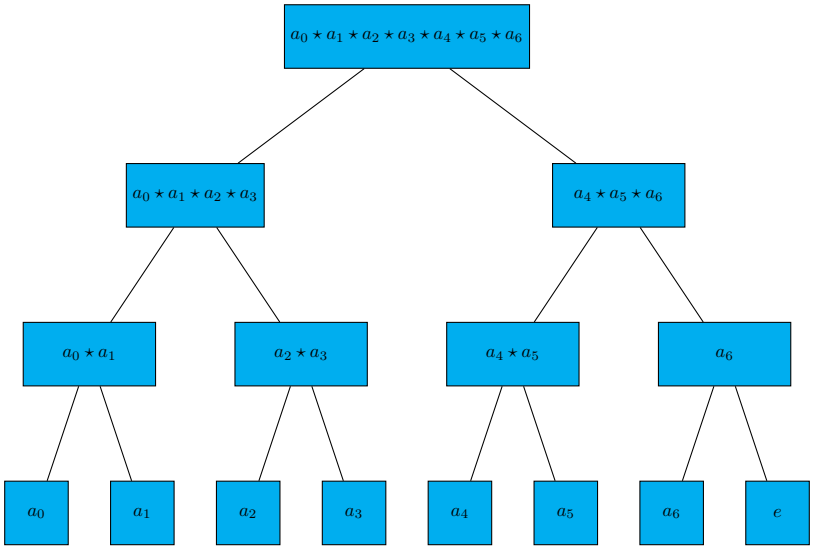
### 1.　セグメント木の構造

セグメント木の構造の特徴は以下の通りです。下の図は $a_0$から$a_6$ の 7 個のデータを持つセグメント木の例です。

- 葉ノードにデータを持つ
- 親ノードは 2 つの子ノードの演算結果を持つ
- $op(x, y)$ は結合法則を満たす $((x * y) * z = x * (y * z)$ が成立する$)$
- 集合 $S$ に対する $op$ に単位元が存在する




図には?があります。?を**全要素の総積をセグメント木の根が持つ**ように埋めていきます。下の図の $e$ は単位元を表しています。根 $a_0 * a_1 * a_2 * a_3 * a_4 * a_5 * a_6$ は全要素の総積になっていることを考えると、その下の? は $a_4 * a_5 * a_6$ になります。同様に、そのほかの?も埋めていくと以下の図のようになります。




### iii.　セグメント木の実装

セグメント木も二分木なので配列で表現することが可能です。ヒープと同様に 1-indexed で実装します。以下の操作ができるようなセグメント木を実装します。

- セグメント木の構築
- 要素の更新
- 区間の演算結果の取得

### 1.　セグメント木の構築

セグメント木では葉にデータが格納され、葉の数はデータの数以上でかつ $2^m$ で表せられる最小の数になります。$2^m$ は 2 進数表記にすると $2^m = 1\underbrace{00\cdots0}_{m}$ となるので、2 進数表記したときに m 桁になる数と $2^m$ になる数であるときは葉の数を $2^m$ にします。つまり、データの数 $n$ が $2^{m-1}+1 \leq n \leq 2^m$ であるとき、葉の数は $2^m$ になります。そして今回のセグメント木は 1-indexed の ed の木になっているため配列の 0 番目の要素は無視します。よって、配列全体の長さは $2m$ 個になります。

セグメント木では扱う演算についての単位元を用意する必要があります。例えば、区間の最大値を求めるセグメント木では、単位元は $-\infty$ になります。区間和では単位元は 0 になります。

葉をデータと単位元で埋めたら、その他のノードは 2 つの子ノードのデータを使って更新していきます。

### 2.　要素の更新

更新クエリに対しては、該当する葉ノードを更新した後、その親ノードを更新していきます。

### 3.　区間の演算結果の取得

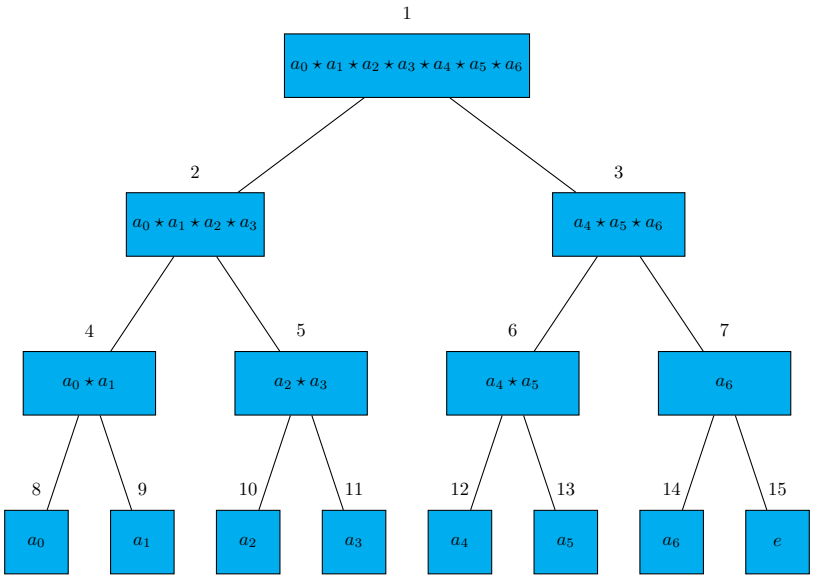
区間の演算結果の取得は少々複雑です。何個か例を挙げて説明します。右開区間 [left, right) の区間の演算結果を求めてみます。下のセグメント木を例にします。ノードの近くの数字は 1-indexed な配列でセグメント木を表現ししたときのインデックス番号です。

・$left = 0, right = 3$ のとき

求めるものは $a_0 * a_1 * a_2 = (a_0 * a_1) * a_1$ つまり 4 と 10 の演算結果になります。

・ $left = 1, right = 7$ のとき

求めるものは $a_0 * a_1 * a_2 * a_3 * a_4 * a_5 * a_6 = a_1 * (a_2 * a_3) * (a_4 * a_5) * a_6$ つまり、9 と 5 と 6 と 14 の演算結果になります。




どのノードを使うかを機械的に判断していくにはどうすればいいでしょうか。それは、$left$ と $right$ が親ノードから見て右子ノードか左子ノードにあるかを見ていけばいいです。left と right が右か左にあるかを見ていき、両者の間隔を狭めていって、left と right が同じノードになるまで続けます。

1. left を右に移動する

left が指すノードを使うか、それともその親ノードを使うかの判定を考えます。left と right の間は連続した区間であることに注意すると、left が親の左側にある場合は親ノードの値を使えばいい

ことがわかります。一方で、left が親の右側にある場合は、left が指すノードの値を使う必要があります。また 1-indexed なセグメント木では右ノードと左ノードは偶奇で判定できます。

2. right を左に移動する

right も同様に考えます。ただ、right は開区間になっているため right - 1 を使うか使わないかを考えます。right が親の左側にある時は right - 1 よりも左側の要素を使うことになりますが、それは right - 1 の親ノードが含んでいるため、right - 1 は使いません。一方、right が親の右側にある場合は right - 1 を使う必要があります。

#### 4. 実装

以上の考察をコードにします。以下は区間の最大値を求めるセグメント木の実装例です。

**コード 6　セグメント木の実装**

```python
# 区間の最大値を求めるセグメント木
class SegmentTree:
    def __init__(self, data_size: int, v: list[int] | None = None):
        self.data_size = data_size
        self.log = (data_size - 1).bit_length()
        self.capacity = 1 << self.log
        self.e = - (1 << 60)
        self.tree = [self.e] * (2 * self.capacity)

        # データが事前に与えられている場合
        if v is not None:
            # 葉ノードを埋める
            for i in range(self.data_size):
                self.tree[self.capacity + i] = v[i]

            # 葉ノード以外を埋める
            for i in range(self.capacity - 1, 0, -1):
                self.update(i)

    def update(self, index: int) -> None:
        self.tree[index] = max(self.tree[2 * index], self.tree[2 * index
            + 1])

    # 更新クエリ
    def set(self, index: int, value: int) -> None:
```

```
25          # 葉に移動
26          index += self.capacity
27          # 更新
28          self.tree[index] = value
29          # 上のノードを更新 根(1)を更新するまで続ける
30          while index > 1:
31              index //= 2
32              self._update(index)
33
34      # 取得クエリ
35      def prod(self, left: int, right: int) -> int:
36          left_value = right_value = self.e
37
38          # 葉に移動
39          left += self.capacity
40          right += self.capacity
41
42          while left < right:
43              # 範囲の左側が右のノード、本実装では奇数
44              if left % 2 == 1:
45                  left_value = max(left_value, self.tree[left])
46                  left += 1
47
48              if right & 2 == 1:
49                  right -= 1
50                  right_value = max(right_value, self.tree[right])
51
52              # 親に移動
53              left //= 2
54              right //= 2
55
56          return max(left_value, right_value)
57
58      def all_prod(self):
59          return self.tree[1]
```

# VII. BIT

BIT(Binary Indexed Tree) はフェニック木とも呼ばれるデータ構造で、要素の値が変化する区間の**和**を高速に求めることができます。セグメント木でも区間和の計算ができますが、BIT はセグメント木よりもシンプルに実装が可能です。区間和に限定すればセグメント木は牛刀をもって鶏を割くようなものです。セグメント木はより一般的な演算に対応したデータ構造です。

BIT はセグメント木に比べてシンプルです。

### i. 和

1 番目から i 番目の要素の和を高速に求めるために、以下のような操作を行います。

1. i を 2 進数表記にしたときに一番右にある 1 の要素を足していく
2. 1 の要素を足していくときに、その要素を取り除いて次の 1 の要素を足していく

例えば、$i = 7$ の場合を考えると、$7 = 111(2)$ となり、右から 1 を引いていくと、$111(2), 110(2), 100(2), 000(2)$、つまり $7, 6, 4, 0$ となります。

BIT のデータを持つ配列を bit とすると、1 から 7 番目の要素の和は、$\mathrm{bit}[0] = 0$ になっているので、

$$\mathrm{bit}[7] + \mathrm{bit}[6] + \mathrm{bit}[4] + \mathrm{bit}[0]$$

になります。

### ii. 更新

要素の更新は和の計算と逆の操作を行います。具体的には、以下の操作を行います。

1. i を 2 進数表記にしたときに一番右にある 1 の要素を引いていく
2. 1 の要素を引いていくときに、その要素を取り除いて次の 1 の要素を引いていく

例えば、$A = [1, 2, 3, 4, 5, 6, 7, 8]$ という等差数列の 5 番目に 6 を足すと、$A = [1, 2, 3, 4, 11, 6, 7, 8]$ になります。このような値の更新を BIT で行ってみましょう。

まず、5 を 2 進数で表すと、$101_2$ となります。ここで、一番右にある 1 の桁に 1 を足す操作を順に行っていきます。$N = 8$ を超える前に操作をやめます。

具体的には、$101_2 \to 110_2 \to 1000_2$ となります。これを 10 進数に直すと、$5 \to 6 \to 8$ となります。ここで出てきた数をインデックスとして、6 を足していくことで値を更新することができます。

実際の操作は以下のようになります。

```
BIT[5] += 6
BIT[6] += 6
BIT[8] += 6
```

### iii. 実装

BIT の実装例は以下の通りです。

**コード 7 BIT の実装**

```python
class BIT:
    def __init__(self, size: int) -> None:
        self.size = size
        self.tree = [0] * (size + 1)

    def sum(self, index: int) -> int:
        res = 0
        while index > 0:
            res += self.tree[index]
            index -= index & (~index + 1)

        return res

    def update(self, index: int, value: int) -> None:
        while index < self.size:
            self.tree[index] += value
            index += index & (~index + 1)
```