پیاده سازی هرم بیشینه $Max - Heap$

آرمان ریاحی

۹ دی ۱۴۰۳

۱ پیاده سازی درخت $Max - Heap$

درخت $Max - Heap$ را با توجه به خصیصه هایی که دارد، به راحتی می توان با لیست پیاده سازی کرد. به طوری که فرزندان چپ و راست هر نود در اندیس iـام، به ترتیب در اندیس های $2i$ـام و $2i+1$ـام قرار می گیرند.

برای ساخت هرم بیشینه یک لیست از اعداد که توالی مشخصی ندارند ابتدا اسکلت درخت را می سازیم که شرایط ساختاری max heap را دارا باشد سپس توالی را سطح به سطح در اسکلت درج می کنیم.سپس از سطح آخر به سمت ریشه و سطح به سطح بررسی می کنیم که آیا گره پدر با فرزندان خود خصیصه max heap بودن را دارد یا خیر. لازم به ذکر است که بررسی برای هر گره تا سطح آخر ادامه پیدا می کند.

Algorithm 1 Max-Heapify(T, i)

Max-Heapify(T, i)
    $left \leftarrow 2i$
    $right \leftarrow 2i + 1$
    $largest \leftarrow i$
    if$(left \leq T.length$ **and** $T[left] > T[i])$ **then**
        $largest \leftarrow left$
    if$(right \leq T.length$ **and** $T[right] > T[largest])$ **then**
        $largest \leftarrow right$
    if$(largest \neq i)$ **then**
        $swap(T[i], T[largest])$
        Max-Heapify$(T, largest)$

الگوریم فوق نود iـام را با فرزندان خود مقایسه میکند و در نهایت از بین سه نود بزرگ ترین نود در جایگاه نود iـام قرار میگیرد(بزرگ ترین نود می تواند خود نود iـام باشد) و این عمل آن قدر تکرار می شود تا نود iـام در بهترین جایگاه قرار گیرد.

برای تعیین پیچیدگی این الگوریتم در بدترین حالت نود iـام باید تا سطح برگ حرکت کند پس پیچیدگی آن به ارتفاع نود بستگی دارد و $O(h)$ است. در بد ترین حالت $O(\log n)$ است.

Algorithm 2 Build-Max-Heap($T[a_1, \ldots, a_n]$)

```
Build-Max-Heap(T)
    for(i from ⌊T.length/2⌋ downto 1) do
        Max-Heapify(T, i)
```

الگوریتم فوق از آخرین نودی که فرزند دارد الگوریتم max heapify را تا ریشه پیاده سازی می کند. در آخر درخت حاصل یک هرم بیشینه است.

برای تحلیل پیچیدگی این الگوریتم می دانیم بیشینه تعداد گره های با ارتفاع h در یک هرم با n عنصر برابر $\lceil \frac{n}{2^{h+1}} \rceil$ است.

از آن جایی که هزینه max heapify برای گره های با ارتفاع h برابر $O(h)$ و $0 \le h \le \lfloor \log n \rfloor$، داریم:

$$T(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \right)$$

از رابطه $\sum_{k=0}^{\infty} \frac{k}{x^k} = \frac{x}{(x-1)^2}$ ، وقتی $|x| > 1$ داریم:

$$\sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} < \sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

در نتیجه:

$$T(n) = O(2n) = O(n)$$

۲ عملیات افزایش یک عنصر در درخت $Max - Heap$

در این عملیات می خواهیم اگر نود iـام مقداری کمتر از $value$ دارد، مقدار آن را به $value$ افزایش دهیم.

Algorithm 3 Increase($T, i, value$)

```
Increase(T, i, value)
    if value < T[i] then
        return error "New value is smaller than current value."
    T[i] ← value
    while(i > 1   and   T[i] > T[⌊i/2⌋]) do
        swap(T[i], T[⌊i/2⌋])
        i ← ⌊i/2⌋
```

در الگوریتم فوق اگر نود iـام مقداری کمتر از $value$ داشته باشد، مقدار آن را به $value$ افزایش می دهیم. سپس با نود پدر مقایسه می شود اگر بزرگ تر بود جا به جا می شود. این روند تا زمانی که به ریشه برسیم یا از نود پدر کوچک تر شود ادامه می یابد. که در بدترین حالت تا ریشه ادامه می یابد پس پیچیدگی آن از مرتبه $O(\log n)$ است.

۳ عملیات درج یک عنصر در درخت $Max - Heap$

Algorithm 4 Insert($T, value$)

```
Incert(T, value)
    T.length ← T.length + 1
    T[T.length] ← −∞
    Increase(T, T.length, value)
```

در الگوریتم فوق نود جدیدی می سازیم و مقداد آن را برابر کوچک ترین عدد ممکن قرار می دهیم. سپس با استفاده از الگوریتم Increase مقدار آن را به $value$ افزایش می دهیم.

با توجه به الگوریتم Increase ، $O(\log n)$ برای افزایش مقدار هزینه می شود و $O(n)$ برای به روز کردن لیست هزینه می شود پس پیچیدگی از مرتبه $O(n)$ است.

۴ عملیات حذف عنصر Max از درخت $Max - Heap$

Algorithm 5 Delete-Max(T)

```
Delete-Max(T)
    if(T.length < 1) then
        return error "Heap is empty."
    maxValue ← T[1]
    T[1] ← T[T.length]
    T.length ← T.length − 1
    Max-Heapify(T, 1)
    return maxValue
```

در الگوریتم فوق آخرین نود را جایگزین ریشه می کنیم و با الگوریتم Max-Heapify درخت را به روز رسانی می کنیم.

با توجه به الگوریتم Max-Heapify ، $O(\log n)$ برای به روز کردن درخت هزینه می شود و $O(n)$ برای به روز کردن لیست هزینه می شود پس پیچیدگی آن $O(n)$ است.

۵ عملیات مرتب سازی با درخت $Max - Heap$

Algorithm 6 Sort-Heap(T)

```
Sort-Heap(T)
    sortHeap = [ ]
    while (T.length > 0) do
        add Delete-Max(T) to sortHeap
    return reverse sortHeap
```

در الگوریتم فوق، با توجه به خصیصه درخت بیشینه می توان ریشه(بزرگ ترین عدد) را حذف و به لیستی افزود و این کار را تا زمانی که درخت تهی شود ادامه داد. بنابراین لیستی مرتب شده به صورت نزولی داریم. در آخر لیست را برعکس می کنیم.

حلقه الگوریتم n بار(به اندازه نود ها) تکرار می شود و در هر تکرار $O(\log n)$ هزینه پرداخت می شود. پس پیچیدگی آن از مرتبه $O(n \log n)$ است.

کد پایتون پیاده سازی هرم بیشینه با توجه به الگوریتم های بالا به شکل زیر است:

```
class MaxHeap:
  def __init__(self):
    self.heap = []

  def max_heapify(self, i):
    left = 2*i + 1
    right = 2*i + 2
    largest = i
    if left < len(self.heap):
      if self.heap[left] > self.heap[i]:
        largest = left
      else:
        largest = i
    if right < len(self.heap) and self.heap[right] > self.heap[largest]:
      largest = right
    if largest != i:
      self.heap[i], self.heap[largest] = self.heap[largest], self.heap[i]
      self.max_heapify(largest)

    def build_max_heap(self, array):
      self.heap = array
      for i in range(len(self.heap) // 2 - 1, -1, -1):
        self.max_heapify(i)

  def increase(self, i, value):
    if value < self.heap[i]:
      raise ValueError("New value is smaller than current value")
    self.heap[i] = value
    while i > 0 and self.heap[i] > self.heap[(i-1) // 2]:
      self.heap[i], self.heap[(i-1) // 2] = self.heap[(i-1) // 2], self.heap[i]
      i = (i-1) // 2

  def insert(self, value):
    self.heap.append(float('-inf'))
    self.increase(self, len(self.heap) - 1, value)

  def delete_max(self):
    if len(self.heap) < 1:
      raise IndexError("Heap is empty.")
    maxValue = self.heap[0]
    self.heap[0] = self.heap[-1]
    self.heap.pop()
    self.max_heapify(0)
    return maxValue

  def sort_heap(self):
    sortHeap = []
    while len(self.heap) > 0:
      sortHeap.append(self.delete_max())
    return sortHeap[::-1]
```