دانشگاه صنعتی شریف
دانشکدهٔ ریاضی

ساختمان‌داده‌ها

بهار ۱۴۰۰

استاد: حسین بومری
گردآورنده:علی سرائر

جلسهٔ پانزدهم

در این جلسه به چند داده‌ساختار می‌پردازیم.

Priority Queue

دستورات Priority Queue به صورت زیر هستند

```
Priority Queue
  - push    //Add a value into Queue
  - pop   //Extract Minimum Value from Queue
  - top   //Get Minimum Value from Queue
  - size    //Size of the Queue

```

به عنوان مثال فرض کنید دستورات زیر اجرا شوند(خروجی هر دستور پس از فلش نشان داده شده است)

```
push 10
push 12
push 4
pop  -> 4
pop  -> 10
push 6
pop  -> 6
pop  -> 12

```

Tree

ابتدا مفهوم درخت را بررسی می‌کنیم.
یکی از کمیت‌هایی که برای درخت تعریف می‌شود، branchFactor است، که به معنی این است که هر node به چند node وصل شده است. در درخت دو-دویی(باینری)، از آنجایی که یک بچه راست داریم و یک بچه چپ و هر node یک پدر دارد،بنابراین به هر node سه node وصل است و branchFactor برابر سه می‌شود.
اگر یادتان باشد، در جلسات گذشته، در پیاده‌سازی کد LinkedList برای هر مؤلفه یک node و یک value تعریف کردیم.[1]
اینجا نیز همین کار را می‌کنیم. برای هر node در درخت یک value تعریف می‌کنیم که نشان دهندهٔ ارزش قرار گرفته در آن node درخت است. همچنین یک لیست تعریف می‌کنیم و در آن node های تمام بچه‌های این node را ذخیره می‌کنیم.
به بالاترین node که هیچ پدری ندارد root و به پایین ترین node ها که هیچ بچه‌ای ندارند leaf یا برگ می‌گوییم.
به node های وسط که هم پدر دارند و بهم بچه نیز internal node یا نود میانی می‌گوییم.
BinaryTree یا درخت دودویی نیز مانند درخت است،با این تفاوت که هر node ماکسیمم دو بچه دارد. یک بچهٔ چپ و یک بچهٔ راست. یک استفاده خوب که می‌توان از درخت دودویی کرد، تعریف کردن بزرگی و کوچکی روی رابطهٔ پدر و بچه‌های یک node مشخص است. مثلا می‌توان BinarySearchTree یا درخت جستجوی دودویی را تعریف کرد. BinarySearchTree به این صورت تعریف می‌شود که value تمام بچه‌های راست هر node از value

[1] البته چون آنجا DoublyLinkedList را بررسی کردیم برای هر node دو node تعریف شد،یکی next و دیگری previous

خودش بزرگ‌تر و value تمام بچه‌های چپ از value خودش کوچک‌تر است.(دقت کنید که برای این تعریف، باید در درخت جستجوی دودویی، یک operator یا عملگر مقایسه‌ای تعریف کنیم)

```
TreeNode
  value
  list(TreeNode)
  parent      //some TreeNodes keep parents some don't, depending on
                //how much space we want to spare
BinaryTreeNode
  value
  rightChild
  leftChild

BinarySearchTree
  Operator <

```

الان می‌توانیم مرتبهٔ زمانی دستورات درخت و درخت دودویی را بررسی کنیم.

```
Tree:
  TreeNode: root
  int: size
  toArray() -> T(n) = T(i1) + T(i2) + ... + T(ik) + O(1) -> O(n)
BinaryTree:
  BinaryTreeNode: root
  int: size
  toArray() ->  T(n) = T(n-i-1)+T(i)+O(1) -> O(n)
  ith_sorted_element() -> O(n)
  getMin()  -> O(n)
  getMax()  -> O(n)
  extractMin()  ->  O(n)
  extractMax()  ->  O(n)
  add()   ->  O(H)
  delete()  -> O(n)
  Build()   -> O(n)   -> We will talk about this a little more in the
                         next session
  decreaseKey()  -> O(1)
  getSorted()   -> O(n log n)

```

مرتبهٔ زمانی دو تابع اول در زیر توضیح داده شده‌اند. بقیه در بخش Heap بررسی می‌شوند.

toArray()

برای تبدیل به آرایه شدن می‌توان دستور را به صورت یک تابع بازگشتی در نظر گرفت؛ به این صورت که هر پدر به بچهٔ چپش دستور می‌دهد که به آرایه تبدیل شوند. پس از آن‌که کل بچه‌های چپ آرایه شدند، خودش درون آرایه قرار می‌گیرد و سپس همین دستور را به بچه‌های راستش می‌دهد.
شکل(۱) را به عنوان مثال در نظر بگیرید. اول node عدد ۴ به ۷ دستور می‌دهد که آرایه شود و آن آرایه را برگرداند. عدد ۷ هم همین‌کار را تکرار می‌کند. به این صورت که به ۹ همین دستور را می‌دهد. پس از آن‌که ۹ تبدیل به آرایه شد، به خود ۷ بازمی‌گردد و خودش نیز درون آرایه قرار می‌گیرد. سپس به ۸ دستور می‌دهد که تبدیل به آرایه شود. پس از آن‌که ۷ و بچه‌هایش به آرایه تبدیل شدند، به ۴ بازمی‌گردد و ۴ در آرایه نوشته می‌شود. سپس به فرزند سمت راست دستور داده می‌شود که آرایه شود و از آن‌جا که بچه‌ای ندارد، خودش در آرایه نوشته می‌شود و به این ترتیب کار این تابع به پایان می‌رسد. برای این مثال، خروجی به صورت زیر می‌شود

۹ ۷ ۸ ۴ ۱۱

اگر دقت کنید متوجه می‌شوید که در طی این فرایند، از روی هر عنصر فقط یک‌بار عبور کردیم(توجه کنید که در مرحلهٔ اول از روی عدد ۴ به عدد ۷ نمی‌رویم، بلکه بلافاصله به فرزند چپش، یعنی ۹ رفتیم و پس از آن به عدد ۷ دستور دادیم که در آرایه نوشته شود). بنابراین مرتبهٔ زمانی این دستور از $O(n)$ است. یک تفسیر دیگری که می‌توان در درخت دودویی انجام داد این است که، در تابع toArray() وقتی که از ۴ به ۷ یا ۱۱ می‌رویم، از آنجایی که هر پدر ماکسیمم دوتا بچه می‌تواند باشد، که مقداری ثابت است، می‌توانیم این صدا زدن ۲ بچه‌را نیز اعمالی در نظر بگیریم که وقتی انجام می‌شود که روی پدر آن‌ها یعنی ۴ قرار داریم، یعنی رفتن به فرزند سمت چپ و راست را روی هزینهٔ مرور و نوشتن خود ۴ روی آرایه در نظر می‌گیریم.(یعنی به ازای هر node داریم از $O(۱)$ کار انجام می‌دهیم). بنابراین، مرتبهٔ زمانی این تابع خطی است.
می‌توان محاسبهٔ مرتبه زمانی آن را به صورت زیر نوشت

$$T(n) = T(n-i-۱) + T(i) + O(۱) \to O(n)$$

که در آن $T(n-i-۱)$ هزینهٔ آرایه شدن بچه‌های چپ و $T(i)$ هزینهٔ آرایه شدن بچه‌های راست و خود پدر است.
توجه: این استدلال برای درخت به طور کلی نیز جواب می‌دهد. تابع toArray() برای درخت دلخواه به صورت زیر می‌شود

$$T(n) = T(i_۱) + T(i_۲) + \cdots + T(i_k) + O(۱) \to O(n) \quad , \sum_{j=۱}^{k} i_j = n$$

ith _ sorted _ element()

برای به‌دست آوردن خروجی این دستور کافی است آرایه‌ای که از دستور toArray() به دست می‌آید را با بهینه‌ترین الگوریتم سورت، که از مرتبهٔ زمانی $O(n\,log\,n)$ است مرتب کنیم و عضو i ام را به عنوان خروجی بدهیم. بنابراین مرتبهٔ زمانی این دستور از مرتبهٔ $O(n) + O(n\,log\,n) \to O(n)$ می‌شود(هزینهٔ $O(n)$ در این محاسبه، هزینهٔ تبدیل به آرایه کردن درخت بود.)
اما یک راه بهتر وجود دارد. می‌توانیم از الگوریتمی که در جلسهٔ سیزدهم برای پیدا کردن میانه گفته شد استفاده کنیم و i امین عنصر مرتب را در زمان $O(n)$ بیابیم. (دقت کنید که از آن الگوریتم نه‌تنها می‌توان برای پیدا کردن میانه، بلکه برای پیدا کردن هر عنصری استفاده کرد)

extractMin()

ابتدا مینیمم را از مرتبهٔ زمانی $O(n)$ پیدا می‌کنیم(به این صورت که درخت را در مرتبهٔ زمانی $O(n)$ تبدیل به آرایه می‌کنیم و کل آن را پیمایش می‌کنیم تا مینیمم را پیدا کنیم.).
سپس آن را حذف می‌کنیم. اما الان باید یک جایگزین برای بچهٔ پدر این node پیدا کنیم. مثلا شکل(۱) را در نظر بگیرید. اگر node مربوط به ۷ را حذف کنیم، باید بچهٔ عدد ۴ را جایگزین کنیم. فرض کنید ۹ و ۸ هر کدام دو بچه داشتند. بنابراین هیچکدام از این دو را نمی‌توانستیم جای ۷ بنشانیم، چرا که به این صورت تعداد بچه‌های این node برابر ۳ می‌شد که در درخت دودویی چنین چیزی ممکن نیست.
برای حذف عنصر می‌توان یک بچه را تا آخر جلو برویم تا به یک برگ برسیم. حالا می‌توانیم این برگ را جای node حذف شده قرار دهیم. به عنوان مثال در شکل (۱) اگر ۷ را حذف کنیم، می‌توانیم آن را با ۹ که یک برگ است جایگزین کنیم. مرتبهٔ زمانی این‌کار برابر مرتبهٔ زمانی پیدا کردن برگ است که برابر $O(H)$ می‌شود. که H عمق درخت است. واضح است که $H < n$ بنابراین مرتبهٔ زمانی کل این تابع نیز همان $O(n)$ است.
یادآوری: تعریف عمق درخت به این صورت است: عمق ریشه برابر صفر است. عمق هر بچه هم برابر عمق پدر به اضافهٔ یک است.

delete()

مانند ()extractMin است با این تفاوت که مکان node ای که می‌خواهیم پاک کنیم را به عنوان ورودی می‌گیریم. بنابراین مرتبهٔ زمانی این تابع از $O(H)$ است.

add()

برای اضافه کردن یک node باید برگ را پیدا کنیم و این node جدید را آن‌جا اضافه کنیم. طبق استدلالی که در توابع قبلی کردیم می‌دانیم این کار در $O(H)$ انجام‌پذیر است.

Build()

از آن‌جا که در درخت ترتیب خاصی برای قرار گرفتن اعداد نداریم، می‌توانیم هر عنصر را add کنیم، با این تفاوت که در اینجا خودمان می‌دانیم که در کجا می‌توانیم عنصر جدید را اضافه کنیم. در نتیجه مانند تابع ()add در بالا نیازی به یافتن برگ‌ها نیست. در نتیجه می‌توان این‌کار را در $O(n)$ زمان انجام داد.

decreaseKey()

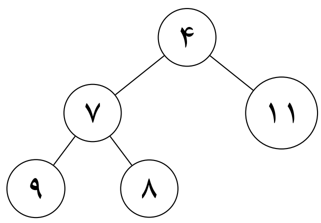
این تابع درخواست می‌کند که value یک node مشخص به اندازه‌ای مشخص کاهش پیدا کند. واضح است که در یک درخت دودویی معمولی که هیچ شرطی روی node های آن وجود ندارد، این کار در $O(۱)$ انجام‌پذیر است.

getSorted()

ابتدا درخت را به آرایه تبدیل می‌کنیم، که از مرتبهٔ $O(n)$ زمان می‌برد. سپس با بهینه‌ترین الگوریتم سورت آن را در $O(n\,log\,n)$ مرتب می‌کنیم. بنابراین مرتبهٔ زمانی کلا $O(n\,log\,n)$ می‌شود.

Heap

حال به بررسی Heap می‌پردازیم. Heap یا پشته یک نوع درخت دودویی است که آن ارزش همه بچه‌های یک node از ارزش خودش همه بزرگ‌تر(یا همه کوچک‌ترند).
به عنوان مثال شکل زیر نشان‌دهندهٔ یک Heap است.



شکل(۱)

دقت کنید که بچه یا بچه‌های هر node از خود آن node همیشه بزرگ‌ترند. مثلا نمی‌توانستیم در خانه‌ای که ۸ قرار دارد عدد ۳ را جای آن قرار دهیم، چرا که عدد ۳ از پدرش که ۷ است کوچک‌تر می‌شود.به این نوع پشته که عضوهای کوچک‌تر بالاتر قرار می‌گیرند(پدرها از بچه‌ها کوچک‌ترند)، MinHeap گفته می‌شود.

```
MinHeap(BinaryTree)
  Operator <
  //Values of all children of a node are bigger than its own value
  getMin()  -> O(1)
  getMax()  -> O(n)
  extractMin()  ->  O()
  extractMax()  ->  O()
  add() ->  O()
  delete()  ->  O()
  Build()   ->  O()
  i_th sorted element ->  O()
  decreaseKey()  -> O()
  getSorted()   -> O()

```

getMin()

از آن‌جا که در MinHeap مینیمم همان root است، واضح است که در $O(۱)$ زمان می‌توان مینیمم را پیدا کرد.

getMax()

برای به دست آوردن ماکسیمم می‌توانیم با استفاده از تابع toArray() که بالاتر توضیح دادیم، MinHeap را تبدیل کنیم که از $O(n)$ می‌شود. سپس مقدار ماکسیمم را با یک‌بار پیمایش کل لیست پیدا کنیم که باز هم از $O(n)$ است.
بنابراین پیدا کردن ماکسیمم کلا از $O(n)$ می‌شود.
در جلسهٔ آینده مرتبهٔ زمانی توابع باقی‌مانده بررسی می‌شوند.