درس ساختمان داده‌ها
دانشکده ریاضی. دانشگاه صنعتی خواجه نصیرالدین طوسی بهار ۹۹
تدریس توسط: حسین جوهری

درخت BST

۱ درخت دودویی جستجو BST

درخت دودویی جستجو یا binary search tree به اختصار BST داده ساختاری درختی است که اعمال داده‌ای مثل جستجو، درج و حذف عناصر را پشتیبانی می‌کند. پیاده‌سازی BST بسیار ساده است و هزینه اعمال مختلف در آن در حالت میانگین خوب است.

تعریف: درخت دودویی جستجو یک درخت دودویی است بطوریکه برچسپهای واقع در زیردرخت چپ هر راس از پرچسپ خود راس کمتر و برچسپهای واقع در زیردرخت سمت راست از برچسپ خود راس بیشترند. به عبارت دیگر، اگر راس v برچسپش x باشد، برچسپهای زیردرخت سمت چپ v همه از x کمترند و برچسپهای زیردرخت سمت راست v همه از x بیشترند. با توجه به این قانون برچسپ تکراری در یک BST وجود ندارد.

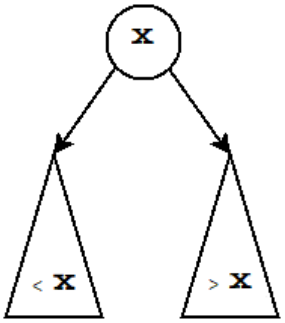
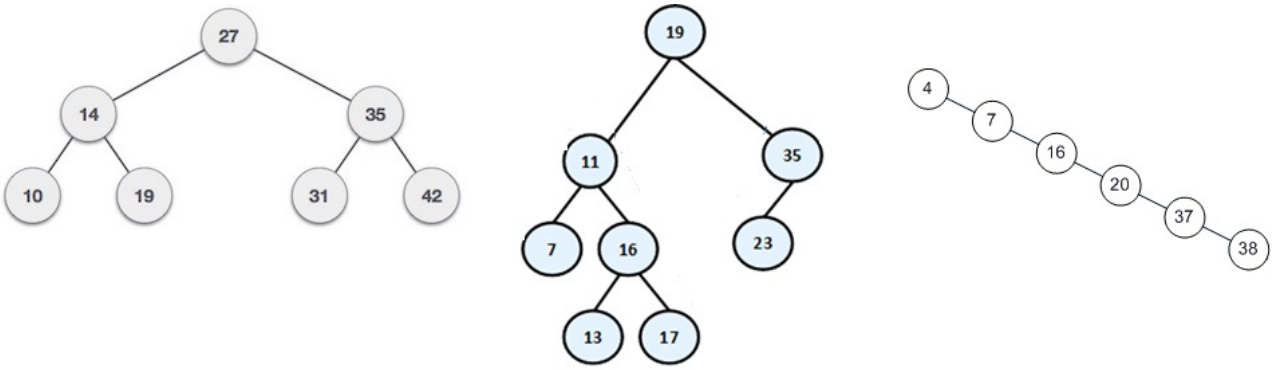


Figure ۱: برچسپهای زیردرخت سمت راست از برچسپ ریشه بزرگترند و برچسپهای زیردرخت سمت چپ از برچسپ ریشه کوچکترند. برای هر راس این قاعده برقرار است.

چند نمونه درخت دودویی جستجو



کمترین ارتفاع یک BST با n عنصر برابر با $\lfloor \log n \rfloor$ است. بیشترین ارتفاع برابر با $n-1$ است.

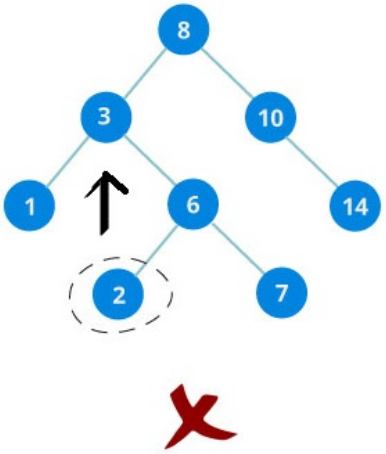


Figure ۲: یک درخت دودویی که BST نیست

۱.۱ ساختارداده BST

برای پیاده‌سازی ساختار داده BST می‌توان از همان شیوه پیاده‌سازی درخت باینری استفاده کرد. اعمال اصلی درخت BST شامل موارد زیر می‌باشد:

- جستجو برای کلید x
- پیدا کردن راس با کلید مینیمم در درخت
- پیدا کردن راس بعدی x: successor(x). در پایین توضیح داده شده است.
- درج راس با کلید جدید در درخت
- حذف راسی که حاوی کلید مینیمم است
- حذف یک راس دلخواه

برای اینکه اعمال بالا زمان اجرای بهتری داشته باشند، برای هر راس درخت، علاوه بر مولفه‌های left و right و element ، مولفه پدر parent را نیز اضافه می‌کنیم تا دسترسی سریع به پدر یک راس فراهم شود. در زیر کلاس Node نشان داده شده است.

```
class Node:
    def __init__(self, left, right, parent, val):
        self.left = left
        self.right = right
        self.parent = parent
        self.element = val

class BSTree:
    def __init__(self):
        self.root = None
...
```

در پایین پیاده‌سازی اعمال بالا ارائه شده است. بعضی از پیاده‌سازی ها را بصورت سطح بالا و شبه کد ارائه کردیم.

۲.۱ جستجو در BST برای کلید x

جستجوی برای کلید x در یک BST الگوریتمی ساده دارد. از ریشه درخت r شروع می‌کنیم، اگر برچسپ ریشه برابر با x بود راس مورد نظر را یافته‌ایم. اگر x کمتر از $r.element$ باشد جستجو را در زیردرخت سمت چپ ریشه ادامه می‌دهیم در غیر اینصورت در زیردرخت سمت راست ریشه دنبال x می‌گردیم. این الگوریتم یک توصیف بازگشتی کوتاه دارد. دقت کنید که اگر کلید x در درخت یافت نشود الگوریتم مقدار None را برمی‌گرداند. اگر x یافت شود اشاره‌گر به راس حاوی کلید x برگردانده می‌شود.

```
BST-Search(r,x):
1. if (r == None) or (x == r.element) : return r
2. if (x < r.element) :
3.       return BST-Search(r.left, x)
4. else:
5.       return BST-Search(r.right, x)
```

همین الگوریتم را می‌توان بصورت غیر بازگشتی پیاده‌سازی کرد.

```
NR-BST-Search(r,x):
1. while (r == None) and (x == r.element):
2.      if (x < r.element):
3.           r = r.left
4.      else:
5.           r = r.right
6. return r
```

مسیر جستجوی x

موقع جستجوی کلید x، به مسیری که در درخت BST طی می‌شود، مسیر جستجوی x گفته می‌شود. به دنباله کلیدهایی که در طول مسیر ملاقات می‌شوند دنباله جستجوی x گفته می‌شود. برای مثال در BST زیر، مسیر جستجوی 14 نشان داده شده است. دنباله

$$19, 11, 16, 13$$

دنباله جستجوی 14 در این BST می‌باشد.

دنباله‌ی $a_1, a_2, \ldots, a_n$ می‌تواند یک دنباله‌ی جستجو باشد اگر و فقط اگر یک BST وجود داشته باشد که $a_1, a_2, \ldots, a_n$ برچسپ رئوس مسیری باشد که از a_1 شروع می‌شود و به a_n ختم می‌شود درحالیکه a_1 ریشه و a_n یک برگ است.

به عبارت دیگر، دنباله‌ی $a_1, a_2, \ldots, a_n$ می‌تواند یک دنباله‌ی جستجو باشد اگر و فقط اگر همه اعداد دنباله متمایز باشند و برای هر a_i در صورتیکه $a_i < a_{i+1}$ همه $a_{i+1}, \ldots, a_n$ از a_i بزرگتر باشند و اگر $a_i > a_{i+1}$ همه $a_{i+1}, \ldots, a_n$ از a_i کوچکتر باشند.

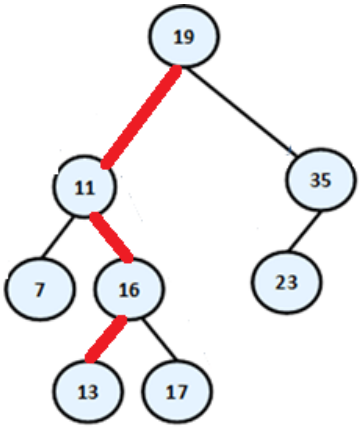


Figure ۳: مسیر جستجوی 14 برجسته شده است

برای مثال دنباله

$$2, 252, 401, 398, 330, 344, 397, 363$$

یک دنباله جستجو است. اما دنباله زیر یک دنباله‌ی جستجو نمی‌تواند باشد چون 912 در زیردرخت سمت چپ 911

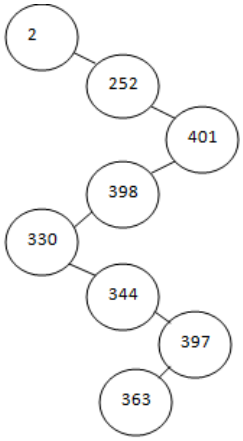


Figure ۴: یک BST متناظر با دنباله جستجوی $2, 252, 401, 398, 330, 344, 397, 363$

قرار می‌گیرد و این تناقض ایجاد می‌کند.

$$925, 202, 911, 240, 912, 245, 363$$

۲ اعمال اصلی برای BST

۱.۲ یافتن عنصر کمینه

عنصر کمینه در BST همیشه در راس انتهای سمت چپ درخت مربوطه قرار دارد. شبه‌کد زیر عنصر کمینه را بصورت بازگشتی پیدا می‌کند. از ریشه شروع می‌کنیم و تا جایی که امکان دارد به سمت چپ می‌رویم. در ابتدای کار r ریشه درخت است.

```
BST-Min(r):
1. if (r.left ==None): return r
2. else:
3.       BST-Min(r.left)
```

۲.۲ یافتن عنصر بعدی successor

successor(x) در واقع کوچکترین عنصر در BST است که از x بزرگتر است. به عبارت دیگر در لیست مرتب شده عناصر successor(x) دقیقا بعد از x قرار می‌گیرد. برای یافتن عنصر بعد از x دو حالت وجود دارد. اگر x زیردرخت سمت راست داشته باشد، کوچکترین عنصر این زیردرخت عنصر بعد از x است.

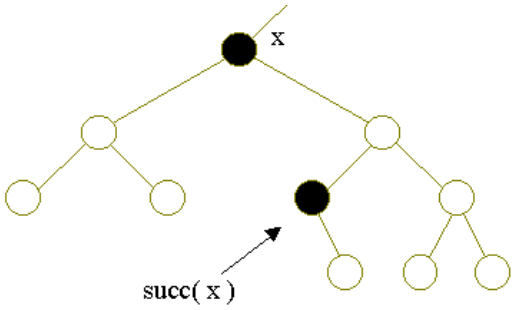


Figure ۵: عنصر مینیمم زیردرخت سمت راست عنصر بعدی x است

اما اگر x زیردرخت سمت راست نداشته باشد، عنصر بعد از x را باید در میان اجداد x جستجو کرد. اولین جد x که x در زیردرخت سمت چپ آن واقع شده است عنصر بعد از x است. پس الگوریتم از راس x به سمت بالا حرکت کرده و اولین بار که به سمت راست برود جد مورد نظر را یافته است.

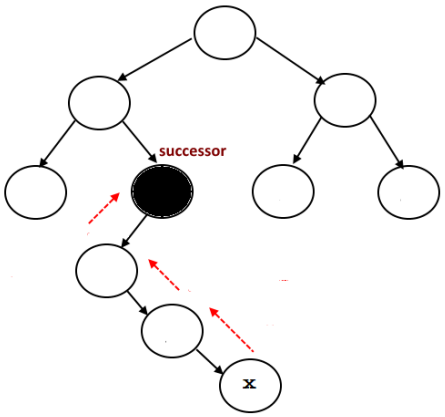


Figure ۶: مثالی که عنصر بعدی x در میان اجداد x یافت می‌شود

اگر جدی برای x با خصوصیات گفته شده یافت نشود، می‌توان نتیجه گرفت که x بزرگترین عنصر BST است (عنصر انتهای سمت راست درخت). شبه کد زیر جزئیات الگوریتم مربوطه را نشان می‌دهد.

```
BST-Successor(x):
1. if (x.right != None):
2.       return BST-Min(x.right)
3. else:
4.       y = x.parent
5. while (y != None and x == y.right):
6.       x = y
7.       y = y.parent
8. return y
```

۳.۲ درج عنصر در BST

برای درج عنصر با کلید x در یک BST فرض ما بر این است که کلید x در BST مورد نظر وجود ندارد. در شبه کد زیر اگر کلید key قبلا وجود داشته باشد، کاری انجام نمی‌شود.

الگوریتم درج در واقع عمل جستجو را برای x انجام می‌دهد. با انجام این کار، محلی از درخت (فرزند چپ یا راست یک برگ) که باید جستجو برای x در آنجا ادامه یابد به عنوان محل درج x انتخاب می‌شود. شبه کد زیر طرز کار این الگوریتم را بصورت بازگشتی بیان کرده است.

```
  def insert(self, key):
       if self.root is None:
           self.root = Node(None, None, None, key)
       else:
           self._insert(self.root,key)

   def _insert(self,r,key):
       if r is not None and r.element < key:
           if r.right is None:
               r.right = Node(None, None, r, key)
           else:
               self._insert(r.right, key)
       elif r is not None and r.element > key:
           if r.left is None:
               r.left = Node(None, None, r, key)
           else:
               self._insert(r.left, key)
```

۴.۲ حذف عنصر کمینه

قبلا دیدیم که عنصر کمینه در یک BST در سمت چپ ترین راس درخت واقع شده است. بدیهی است که این عنصر فرزند سمت چپ ندارد. اما ممکن است فرزند سمت راست داشته باشد. برای حذف عنصر کمینه (در صورت وجود) فرزند راست آن به پدر عنصر حذف شده منتسب می‌شود.

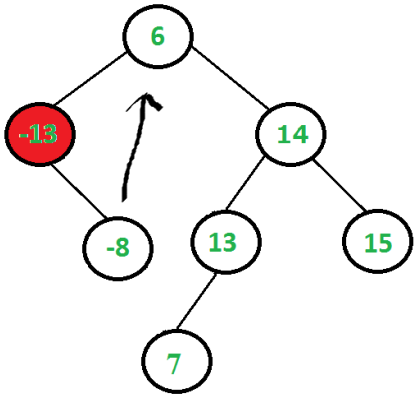


Figure ۷: عنصر کمینه در درخت مشخص شده است. بعد از حذف فرزند راست عنصر حذف شده به پدر عنصر حذف شده منتسب می‌شود

شبه کد زیر طرز کار این الگوریتم را بصورت بازگشتی بیان می‌کند.

```
BST-DeleteMin(r)
1. if(r == None):
2.       print error: tree is empty
3. if(r.left == None):
4.            x = r.element
5.            temp = r
6.            r = r.right
7.            if (r is not None): r.parent = temp.parent
8.            temp.parent.left = r
9.            return x
10. else:
11.          return  BST-DeleteMin(r.left)
```

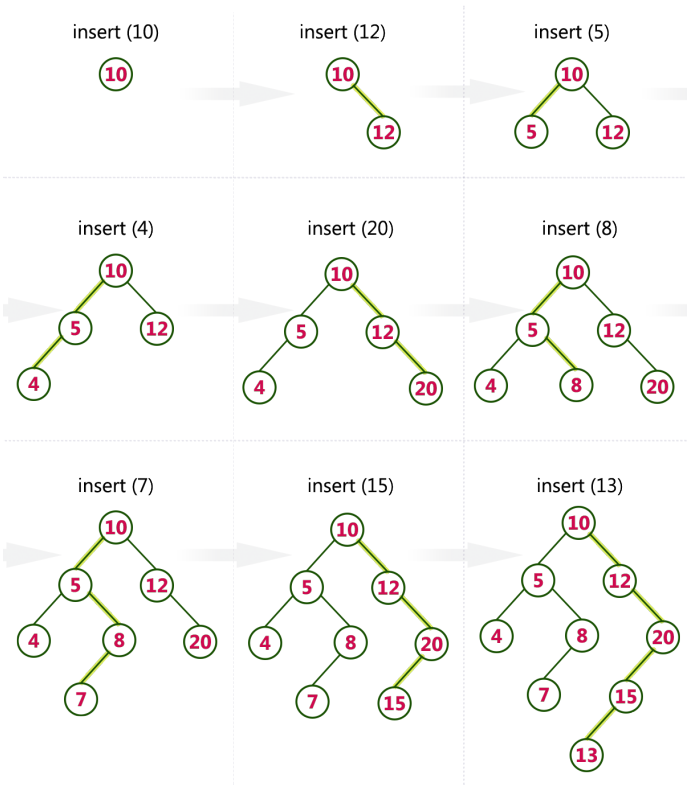
۵.۲ حذف عنصر

برای حذف یک راس با کلید x اگر راس مربوطه فرزند چپ نداشته باشد، فرزند راست جای راس مورد نظر را می‌گیرد. اگر فرزند راست نداشته باشد، فرزند سمت چپ جای آن را می‌گیرد. اگر راس با کلید x هیچ فرزندی نداشته باشد راس مورد نظر (یک برگ) از درخت حذف می‌شود. اگر راس با کلید x هر دو فرزند سمت چپ و راست را داشته باشد، آنگاه عنصر کمینه در زیردرخت سمت راست آن جایگزینش می‌شود. در واقع x جای خود را به successor(x) می‌دهد و successor(x) از درخت حذف می‌شود. حذف successor بدون مشکل خواهد بود چون حداقل فرزند سمت چپ ندارد. در شبه کد زیر ابتدا محل راس مورد نظر با کلید x پیدا می‌شود و سپس پروسه حذف شروع می‌شود.

```
BST-Delete(r,x)
1. if(r == None)
2.      print error: tree is empty
3. if(x < r.element)
4.          BST-Delete(r.left,x)
5. if(x > r.element)
6.          BST-Delete(r.right,x)
7. if(x == r.element)
8.          temp = r
9.          if (r.left == None):
10.                 r = r.right
11.                 r.parent = temp.parent
12.         else if (r.right ==None):
13.                 r = r.left
14.                 r.parent = temp.parent
15.         else
16.                 r.element = BST-DeleteMin(r.right)
```

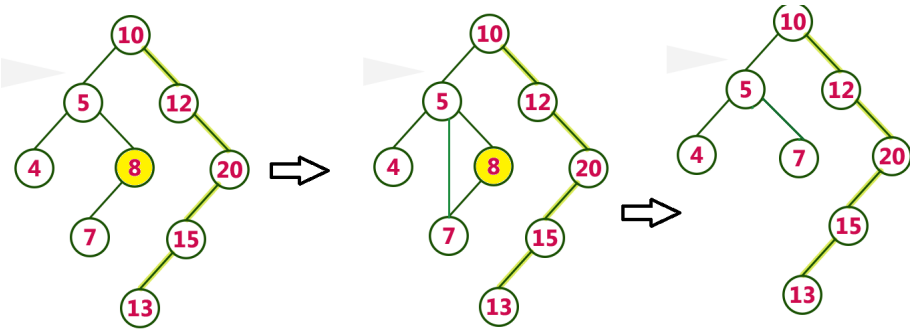
مثالی برای درج و حذف از BST

دنباله درجهای زیر را در یک درخت BST تهی داریم. $10, 12, 5, 4, 20, 8, 7, 15, 13$

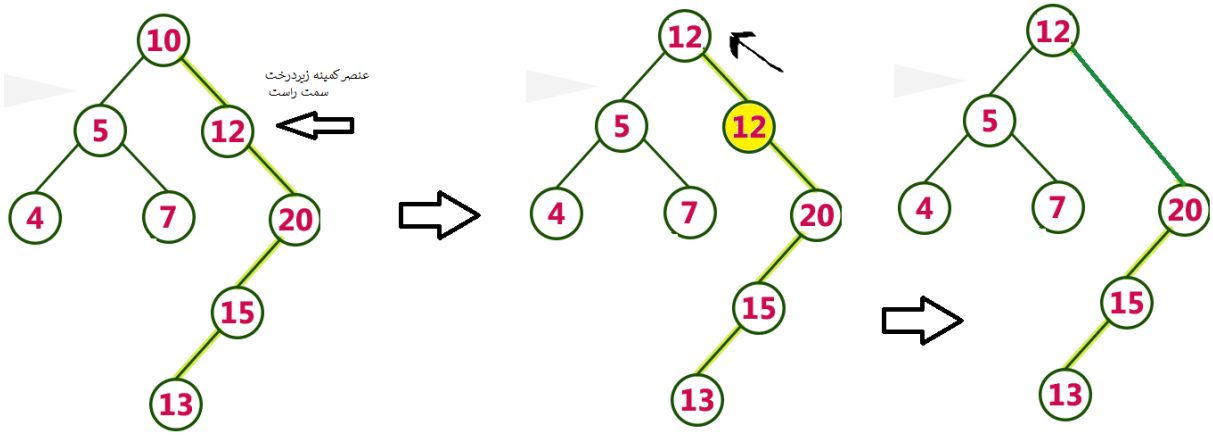


حال فرض کنید 8 را می‌خواهیم حذف کنیم. چون 8 فرزند راست ندارد و فرزند چپ دارد، عنصر 7 به پدر 8 که 5 باشد متنسب می‌شود.

حال فرض کنید 10 را می‌خواهیم حذف کنیم. چون 10 هر دو فرزند چپ و راست را دارد باید عنصر کمینه



زیردرخت سمت راستش را حذف کنیم و با آن جایگزین کنیم.



۶.۲ تحلیل زمان اجرای اعمال اصلی روی درخت BST

همه اعمال بررسی شده در این درس از قبیل یافتن عنصر کمینه، درج و حذف پیچیدگی زمانی‌شان متناسب با ارتفاع درخت BST می‌باشد. با فرض اینکه ارتفاع درخت مورد نظر h است، در همه موارد در بدترین حالت $\Theta(h)$ عملیات اصلی انجام می‌شود.

۷.۲ مرتب‌سازی با استفاده از BST

عناصر یک درخت BST را می‌توان با استفاده از روش پیمایش میان ترتیب inorder بصورت مرتب چاپ کرد.

```
BST-Print-Sorted(r)
1. if(r != None)
2.       BST-Print-Sorted(r.left)
3.       print r.element
4.       BST-Print-Sorted(r.right)
```

از رویه بالا نتیجه می‌شود که می‌توان از ساختار داده BST به عنوان یک ابزار برای مرتب سازی استفاده کرد. عناصر یک آرایه نامرتب را یک به یک در یک ساختار داده‌ی BST درج می‌کنیم و سپس با استفاده از رویه بالا آنها را در آرایه پشت سر هم قرار می‌دهیم. در بدترین حالت این الگوریتم $\Theta(n^2)$ مقایسه انجام می‌دهد چون درج هر

عنصر در BST هزینه‌اش متناسب با ارتفاع درخت است که می‌تواند در حد $\Theta(n)$ بالا رود. موقعی که آرایه ورودی مرتب است (یا برعکس مرتب) ارتفاع درخت با هر درج یک واحد افزایش می‌یابد. درنتیجه در کل $\Theta(n^2)$ مقایسه انجام می‌شود. در بهترین حالت مرتب سازی به $\Theta(n \log n)$ زمان نیاز دارد.