برای پیمایشهایی هستیم که به صورت بازگشتی تعریف می‌شوند. پشته یک ساختمان طبیعی برای پیاده‌سازی به این صورت می‌باشد. بحث و بررسی الگوریتم‌های پشته‌ای برای این منظور، در بخش بعد گنجانده شده است.

## ۵ـ۷ الگوریتم‌های پیمایش به کمک پشته‌ها

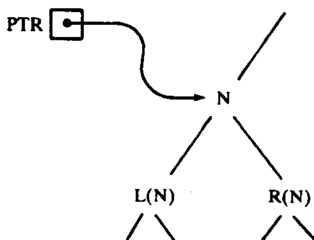
فرض کنید درخت دودویی T به وسیلهٔ نمایش پیوندی در حافظه نگهداری می‌شود :

**TREE(INFO, LEFT, RIGHT, ROOT)**

این بخش پیاده‌سازی سه پیمایش استاندارد T را توضیح می‌دهد که در بخش گذشته به وسیلهٔ زیربرنامه‌های غیربازگشتی و با استفاده از پشته‌ها به صورت بازگشتی تعریف شده بود. سه پیمایش را به صورت مستقل مورد بحث و بررسی قرار می‌دهیم.

### پیمایش PreOrder

الگوریتم پیمایش **PreOrder** از یک متغیر (اشاره‌گر) **PTR** استفاده می‌کند که حاوی مکان گرهٔ **N** ای است که درحال حاضر جستجو و خوانده شد. این‌وضعیت درشکل ۱۵ـ۷ به تصویر درآمده است که در آن **L(N)** بچهٔ چپ گرهٔ **N** و **R(N)** بچهٔ راست گرهٔ **N** را نشان می‌دهد. الگوریتم از آرایهٔ **STACK** نیز استفاده می‌کند، که آدرس گره‌ها را برای پردازش بعدی نگه می‌دارد.




شکل ۱۵ـ۷

**الگوریتم** : در آغاز کار **NULL** را به داخل **STACK**، **push** کنید و بدنبال آن قرار دهید **PTR := ROOT** آنگاه مراحل زیر را تکرار کنید تا وقتی که **PTR = NULL** یا معادل آن **PTR ≠ NULL** است.

**(الف)** کار پردازش را در سمت چپ‌ترین میسر از ریشهٔ **PTR** به طرف پائین انجام می‌دهیم، هر گرهٔ **N** در طول مسیر را پردازش کرده و بچهٔ راست **R(N)** را در صورت وجود، به داخل **STACK**، **Push** می‌کنیم. پس ازآن که برای یک گرهٔ **N** هیچ بچهٔ چپ **L(N)** پردازش نشده موجود نباشد پیمایش به پایان می‌رسد.

بـنابرایـن PTR بـا اسـتفاده از دسـتور جـایگزینی PTR := LEFT[PTR] تـازه مـی‌شود و وقـتی LEFT[PTR] = NULL است پیمایش متوقف می‌شود.

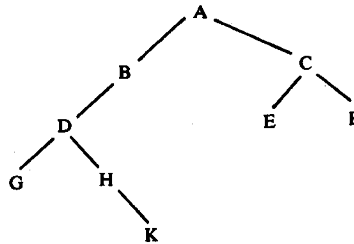
(ب) [بازگشت به عقب BackTracking] . POP کنید و در PTR عنصر بالای پشته STACK را جایگزین کنید. اگر PTR ≠ NULL ، آنگاه به مرحلهٔ (الف) برگردید، درغیر اینصورت کار به پایان می‌رسد Exit .

یادآور می‌شویم که عنصر اولیه NULL در پشتهٔ STACK به صورت یک نگهبان مورد استفاده قرار گرفته است.

این الگوریتم را در مثال بعد شبیه‌سازی می‌کنیم. اگرچه مثال با خود گره‌ها کار می‌کند اما در کاربردهای عملی، مکان گره‌ها در PTR جایگزین می‌شود و به داخل پشتهٔ STACK ، Push می‌شود.

## مثال ۹ـ ۷

درخت دودویی T شکل ۱۶ـ ۷ را درنظر بگیرید.




شکل ۱۶ـ ۷

الگوریتم بالا را با درخت T شبیه‌سازی می‌کنیم، محتوای STACK را در هر مرحله نشان می‌دهیم.

۱ـ NULL را در آغاز به داخل STACK ، Push کنید. STACK : Φ .

آنگاه قرار دهید PTR := A ، که ریشهٔ درخت T است.

۲ـ از سمت چپ‌ترین مسیر ریشهٔ PTR = A به طرف پائین و به شرح زیر پیش بروید.

(i) A را پردازش کنید و بچهٔ سمت راست آن، C را به داخل پشتهٔ STACK ، Push کنید. STACK : Φ, C .

(ii) B را پردازش کنید. هیچ بچهٔ راست وجود ندارد.

(iii) D را پردازش کنید و بچهٔ راست آن، H را به داخل پشتهٔ STACK ، Push کنید.

STACK: Φ, C, H

(iv) G را پردازش کنید. هیچ بچهٔ راست وجود ندارد.

از آنجا که G بچهٔ چپ ندارد، هیچ گرهٔ دیگری پردازش نمی‌شود.

۳ـ [بازگشت به عقب BackTracking.] عنصر بالا H را از پشتهٔ STACK ، pop کنید و قرار دهید PTR := H.
درنتیجه :

STACK: Φ, C

چون PTR ≠ NULL ، به مرحلهٔ (الف) الگوریتم برگردید.

۴ـ از سمت چپ‌ترین مسیر ریشهٔ PTR = H به طرف پائین و به شرح زیر پیش بروید.

(v) H را پردازش کنید و بچهٔ راست آن K ، را به داخل پشتهٔ STACK ، Push کنید : از آنجا که H بچهٔ چپ ندارد. هیچ گرهٔ پردازش نشدهٔ دیگری وجود ندارد،

STACK: Φ, C, K

۵ـ [بازگشت به عقب BackTracking.] K را از پشتهٔ STACK ، pop کنید و قرار دهید PTR := K درنتیجه:

STACK: Φ, C

چون PTR ≠ NULL ، به مرحلهٔ (الف) الگوریتم برگردید.

۶ـ از سمت چپ‌ترین مسیر ریشهٔ PTR = K به طرف پائین و به شرح زیر پیش بروید.

(vi) K را پردازش کنید. بچهٔ راست وجود ندارد.

از آنجا که K بچهٔ چپ ندارد، هیچ گرهٔ پردازش نشدهٔ دیگری وجود ندارد.

۷ـ [بازگشت به عقب BackTracking.] K را از پشتهٔ STACK ، pop کنید و قرار دهید PTR := C
درنتیجه :

STACK: Φ

چون PTR ≠ NULL به مرحلهٔ (الف) الگوریتم برگردید.

۸ـ از سمت چپ‌ترین مسیر ریشهٔ PTR = C به طرف پائین و به شرح زیر پیش بروید :

(vii) C را پردازش کنید و بچه راست آن F را به داخل پشتهٔ STACK ، Push کنید :

STACK: Φ, F

(viii) E را پردازش کنید. بچهٔ راست وجود ندارد.

۹ـ [بازگشت به عقب BackTracking.] F را از پشتهٔ STACK ، pop کنید و قرار دهید PTR := F درنتیجه:

STACK: Φ

قرار دهید PTR ≠ NULL به مرحلهٔ (الف) الگوریتم برگردید.

۱۰ـ از سمت چپ‌ترین مسیر ریشهٔ PTR = F به طرف پائین و به شرح زیر پیش بروید :

(ix) F را پردازش کنید. بچهٔ راست وجود ندارد.

از آنجا که F بچهٔ چپ ندارد، هیچ گرهٔ پردازش‌نشدهٔ دیگری وجود ندارد.

۱۱- [بازگشت به عقب BackTracking.] عنصر بالا NULL را از پشتهٔ STACK ، pop کنید و قرار دهید PTR := NULL چون PTR = NULL الگوریتم به پایان رسیده است.

همانگونه که از مرحله‌های ۲ ، ۴ ، ۶ ، ۸ ، ۱۰ نتیجه می‌شود. گره‌های پردازش‌شده به ترتیب A, B, D, G, H, K, C, E, F هستند. این همان پیمایش PreOrder موردنظر درخت T است.

نمایش رسمی الگوریتم پیمایش preOrder به صورت زیر است :

```
Algorithm 7.1: PREORD(INFO, LEFT, RIGHT, ROOT)
  A binary tree T is in memory. The algorithm does a preorder traversal of T,
  applying an operation PROCESS to each of its nodes. An array STACK is used
  to temporarily hold the addresses of nodes.

  1. [Initially push NULL onto STACK, and initialize PTR.]
     Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
  2. Repeat Steps 3 to 5 while PTR ≠ NULL:
  3.     Apply PROCESS to INFO[PTR].
  4.     [Right child?]
         If RIGHT[PTR] ≠ NULL, then: [Push on STACK.]
             Set TOP := TOP + 1, and STACK[TOP] := RIGHT[PTR].
         [End of If structure.]
  5.     [Left child?]
         If LEFT[PTR] ≠ NULL, then:
             Set PTR := LEFT[PTR].
         Else: [Pop from STACK.]
             Set PTR := STACK[TOP] and TOP := TOP - 1.
         [End of If structure.]
     [End of Step 2 loop.]
  6. Exit.
```

## پیمایش InOrder

الگوریتم پیمایش InOrder نیز از متغیر اشاره‌گر PTR استفاده می‌کند که حاوی مکان گرهٔ N ای است که به‌تازگی جستجو و خوانده شد و نیز از یک آرایهٔ STACK استفاده می‌کند که آدرس گره‌ها را برای پردازش بعدی نگه می‌دارد. درواقع، با این الگوریتم یک گره تنها وقتی پردازش می‌شود که از STACK ، pop شود.

**الگوریتم** : در آغاز کار NULL را به داخل STACK (به عنوان نگهبان) Push کنید و آنگاه قرار دهید PTR := ROOT آنگاه مراحل زیر را تکرار کنید تا وقتی که NULL از STACK ، pop شود.

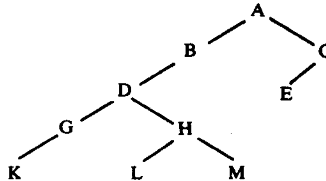
**(الف)** از سمت چپ‌ترین مسیر ریشهٔ PTR به طرف پائین پیش بروید، هر گرهٔ N را به داخل STACK ، push کنید و این کار را هنگامی متوقف کنید تا گرهٔ N هیچ بچهٔ چپ جهت Push شدن در STACK نداشته باشد.

(ب) [بازگشت به عقب BackTracking.] pop کنید و گره‌های STACK را پردازش کنید. اگر NULL ، pop شد آنگاه از الگوریتم خارج شوید Exit . اگر بخواهید گرهٔ N با یک بچهٔ راست R(N) را پردازش کنید قرار دهید PTR = R(N) (با دستور جایگزینی [PTR := RIGHT[PTR) و به مرحلهٔ (الف) برگردید.

تأکید می‌کنیم که گرهٔ N تنها وقتی پردازش می‌شود که از STACK ، pop شده باشد.

## مثال ۱۰ـ۷

درخت دودویی شکل ۱۷ـ۷ را درنظر بگیرید.




شکل ۱۷ـ۷

الگوریتم بالا را با T شبیه‌سازی می‌کنیم و محتوای STACK را نشان می‌دهیم.

۱ـ در آغاز کار NULL را به داخل STACK ، Push کنید :

STACK: $\Phi$

آنگاه قرار دهید PTR := A که ریشهٔ درخت T است.

۲ـ از سمت چپ‌ترین مسیر ریشهٔ PTR = A به طرف پائین پیش بروید، گره‌های G, D, B, A و K را به داخل STACK ، Push کنید:

STACK: $\Phi$, A, B, D, G, K

از آنجا که K بچهٔ چپ ندارد، هیچ گرهٔ دیگری به داخل STACK ، Push نمی‌شود.

۳ـ [بازگشت به عقب BackTracking.] گره‌های G, K و D ، pop شده پردازش می‌شوند. درنتیجه :

STACK: $\Phi$, A, B

چون D بچهٔ راست ندارد، پردازش را در D متوقف می‌کنیم. آنگاه قرار دهید PTR := H که بچهٔ راست D است.

۴ـ از سمت چپ‌ترین مسیر ریشهٔ PTR = H به طرف پائین پیش بروید و گره‌های H و L را به داخل

STACK ، Push کنید.

از آنجا که L بچهٔ چپ ندارد. هیچ گرهٔ دیگری وجود ندارد که به داخل STACK ، Push نشده باشد.

STACK: Φ, A, B, H, L

۵- [بازگشت به عقب BackTracking.] گره‌های L و H ، pop شده و پردازش می‌شود. درنتیجه :

STACK: Φ, A, B

از آنجا که H بچهٔ راست دارد، پردازش را در H متوقف می‌کنیم. آنگاه قرار دهید PTR := M که بچهٔ راست H است.

۶- از سمت چپ‌ترین مسیر ریشهٔ PTR = M به طرف پائین پیش بروید، گرهٔ M را به داخل STACK ، Push کنید.

STACK; Φ, A, B, M

چون M بچهٔ چپ ندارد، هیچ گرهٔ دیگری به داخل STACK ، Push نمی‌شود.

۷- [بازگشت به عقب BackTracking.] گره‌های M ، B و A ، pop شده پردازش می‌شوند. درنتیجه :

STACK; Φ

چون A بچهٔ راست دارد، هیچ عنصر دیگری از STACK ، pop نمی‌شود. قرار دهید PTR := C ، که بچهٔ راست A است.

۸- از سمت چپ‌ترین مسیر ریشهٔ PTR = C به طرف پائین پیش بروید، گره‌های C و E را داخل STACK ، Push کنید.

STACK: Φ, C, E

۹- [بازگشت به عقب BackTracking.] گرهٔ E ، pop شده پردازش می‌شود. چون E هیچ بچهٔ راست ندارد، گرهٔ C ، pop شده و پردازش می‌شود. از آنجا که C بچهٔ راست ندارد، عنصر بعدی یعنی NULL ، از STACK ، pop می‌شود.

الگوریتم اکنون به پایان رسیده است، چون NULL از STACK ، pop شده است. همانگونه که از مرحله‌های ۳ ، ۵ ، ۷ و ۹ ملاحظه می‌شود گره‌ها با ترتیب C, E, A, B, M, H, L, D, G, K پردازش می‌شوند. این همان پیمایش InOrder موردنظرِ درخت دودویی T است.

نمایش رسمی الگوریتم پیمایش InOrder به صورت زیر است :

```
Algorithm 7.2: INORD(INFO, LEFT, RIGHT, ROOT)
    A binary tree is in memory. This algorithm does an inorder traversal of T,
    applying an operation PROCESS to each of its nodes. An array STACK is used
    to temporarily hold the addresses of nodes.

    1. [Push NULL onto STACK and initialize PTR.]
       Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
    2. Repeat while PTR ≠ NULL: [Pushes left-most path onto STACK.]
           (a) Set TOP := TOP + 1 and STACK[TOP] := PTR. [Saves node.]
           (b) Set PTR := LEFT[PTR]. [Updates PTR.]
       [End of loop.]
    3. Set PTR := STACK[TOP] and TOP := TOP - 1. [Pops node from STACK.]
    4. Repeat Steps 5 to 7 while PTR ≠ NULL: [Backtracking.]
    5.     Apply PROCESS to INFO[PTR].
    6.     [Right child?] If RIGHT[PTR] ≠ NULL, then:
               (a) Set PTR := RIGHT[PTR].
               (b) Go to Step 3.
           [End of If structure.]
    7.     Set PTR := STACK[TOP] and TOP := TOP - 1. [Pops node.]
       [End of Step 4 loop.]
    8. Exit.
```

## پیمایش PostOrder

الگوریتم پیمایش PostOrder اندکی پیچیده‌تر از دو الگوریتم قبلی است، زیرا در اینجا لازم است گرهٔ N را در دو وضعیت مختلف ذخیره کنیم. مابین این دو حالت ازطریق Push کردن N یا N – به داخل STACK تمایز قائل می‌شویم، درکاربردهای عملی، مکان N داخل STACK ، Push می‌شود، از این‌رو N – دارای معنی واضحی است. مجدداً از متغیر (اشاره‌گر) PTR که حاوی مکان گرهٔ N است استفاده می‌شود که اخیراً خوانده شده است، این وضعیت در شکل ۱۵ـ۷ نشان داده شده است.

**الگوریتم** : در آغاز کار NULL را به داخل STACK به عنوان نگهبان Push کنید و آنگاه قرار دهید PTR := ROOT بدنبال آن مراحل زیر را تکرار کنید تا NULL از STACK ، pop شود.

**(الف)** از سمت چپ‌ترین مسیر ریشهٔ PTR به طرف پائین پیش بروید. در هر گره N از مسیر، N را به داخل STACK ، Push کنید و اگر N بچهٔ راست R(N) دارد، R(N) – را به داخل STACK ، Push کنید.

**(ب) [بازگشت به عقب .BackTracking]** pop کنید و گره‌های مثبت داخل STACK را پردازش کنید. اگر NULL ، pop شد، آنگاه از الگوریتم خارج شوید Exit . اگر یک گرهٔ منفی pop شد، یعنی اگر به‌ازای بعضی گرهٔ N، N ـ = PTR، قراردهید PTR = N (با جایگزینی PTR := – PTR) و به مرحلهٔ (الف) برگردید.

تأکید می‌کنیم که گرهٔ N تنها وقتی پردازش می‌شود که از STACK ، pop شود و مثبت باشد.

**مثال ۱۱ـ ۷**

بار دیگر درخت دودویی شکل ۱۷ـ ۷ را درنظر بگیرید. الگوریتم بالا را با **T** شبیه‌سازی می‌کنیم و محتوای **STACK** را نشان می‌دهیم.

۱ـ در آغاز **NULL** را به داخل **STACK**، **Push** کنید و قرار دهید **PTR := A** که ریشهٔ **T** است.

**STACK: Φ**

۲ـ از سمت چپ‌ترین مسیر ریشهٔ **PTR = A** به طرف پائین پیش بروید، گره‌های **G, D, B, A** و **K** را به داخل **STACK**، **Push** کنید. علاوه براین، چون **A** بچهٔ راست **C** دارید، **C−** را پس از **A** اما قبل از **B** به داخل **STACK**، **Push** کنید و چون **D** بچهٔ راست **H** دارد، **H−** را پس از **D** اما قبل از **G** به داخل **STACK**، **Push** کنید. درنتیجه :

**STACK: Φ, A, − C, B, D, − H, G, K**

۳ـ **[بازگشت به عقب BackTracking.]** **pop** کنید و **K** را پردازش کنید همچنین **pop** کنید و **G** پردازش کنید. چون **H−** منفی است، تنها **H−**، **pop** می‌شود. درنتیجه :

**STACK: Φ, A, − C, B, D**

اکنون **PTR = − H**. مجدداً قرار دهید **PTR = H** و به مرحلهٔ (الف) برگردید.

۴ـ از سمت چپ‌ترین مسیر ریشهٔ **PTR = H** به طرف پائین پیش بروید، نخست **H** را به داخل **STACK**، **Push** کنید. چون **H** بچهٔ راست **M** دارد، **M−** را پس از **H** داخل **STACK**، **Push** کنید. بالاخره، **L** را داخل **STACK**، **Push** کنید. درنتیجه :

**STACK; Φ, A, − C, B, D, H, − M, L**

۵ـ **[بازگشت به عقب BackTracking.]** **pop** کنید و **L** را پردازش کنید، اما فقط **M−** را **pop** کنید. درنتیجه:

**STACK: Φ, A, − C, B, D, H**

حال **PTR:= − M**. قرار دهید **PTR = M** و به مرحلهٔ (الف) برگردید.

۶ـ از سمت چپ‌ترین مسیر ریشهٔ **PTR = M** به طرف پائین پیش بروید. اکنون، تنها **M** را داخل **STACK**، **Push** کنید. درنتیجه :

**STACK: Φ, A, − C, B, D, H, M**

۷ـ **[بازگشت به عقب BackTracking.]** **pop** کنید و **D, H, M** و **B** را پردازش کنید اما فقط **C −** را **pop** کنید. درنتیجه :

**STACK: Φ, A**

حال PTR= − C قرار دهید PTR = C و به مرحلهٔ (الف) برگردید.

۸ـ از سمت چپ‌ترین مسیر ریشهٔ PTR = C به طرف پائین پیش بروید. نخست C به داخل STACK، Push می‌شود و آنگاه E، درنتیجه :

STACK: $\phi$, A, C, E

۹- [بازگشت به عقب .BackTracking] pop کنید و E، C و A را پردازش کنید. وقتی NULL، pop می‌شود، STACK خالی است و الگوریتم کامل می‌شود.

همانگونه که از مراحل ۳، ۵، ۷ و ۹ ملاحظه می‌شود، گره‌ها با ترتیب A, C, E, B, D, H, M, L, G, K پردازش می‌شوند. این همان پیمایش PostOrder موردنظر درخت دودویی T است.

نمایش رسمی الگوریتم پیمایش Postorder به شرح زیر است:

**Algorithm 7.3:** POSTORD(INFO, LEFT, RIGHT, ROOT)
A binary tree T is in memory. This algorithm does a postorder traversal of T, applying an operation PROCESS to each of its nodes. An array STACK is used to temporarily hold the addresses of nodes.

```
1.  [Push NULL onto STACK and initialize PTR.]
    Set TOP := 1, STACK[1] := NULL and PTR := ROOT.
2.  [Push left-most path onto STACK.]
    Repeat Steps 3 to 5 while PTR ≠ NULL:
3.      Set TOP := TOP + 1 and STACK[TOP] := PTR.
        [Pushes PTR on STACK.]
4.      If RIGHT[PTR] ≠ NULL, then: [Push on STACK.]
            Set TOP := TOP + 1 and STACK[TOP] := −RIGHT[PTR].
        [End of If structure.]
5.      Set PTR := LEFT[PTR]. [Updates pointer PTR.]
    [End of Step 2 loop.]
6.  Set PTR := STACK[TOP] and TOP := TOP − 1.
    [Pops node from STACK.]
7.  Repeat while PTR > 0:
        (a) Apply PROCESS to INFO[PTR].
        (b) Set PTR := STACK[TOP] and TOP := TOP − 1.
            [Pops node from STACK.]
    [End of loop.]
8.  If PTR < 0, then:
        (a) Set PTR := −PTR.
        (b) Go to Step 2.
    [End of If structure.]
9.  Exit.
```

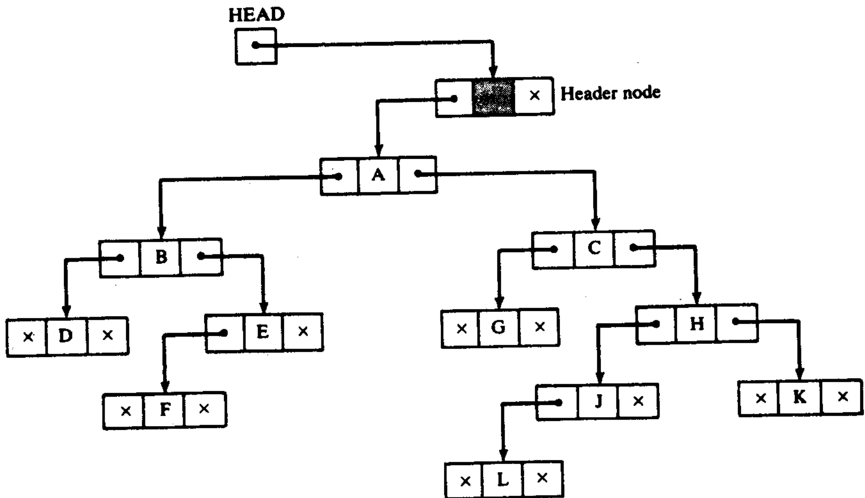
## ۶ـ۷ سرگره‌ها، گره‌های نخ‌کشی شده

درخت دودویی T را درنظر بگیرید. اغلب انواع مختلف از نمایش پیوندی درخت T مورد استفاده

قرار می‌گیرد. زیرا برخی از عملیات اصلاح شده روی T، به منظور پیاده‌سازی ساده‌تر می‌باشند. برخی از این انواع نمایش که مشابه لیستهای دارای سرگره و لیستهای پیوندی چرخشی یا حلقوی است در این بخش مورد بحث و بررسی قرار می‌گیرد.

## سرگره‌ها

درخت دودویی T را درنظر بگیرید که به وسیلهٔ یک نمایش پیوندی در حافظه نگهداری می‌شود. گاهی اوقات گرهٔ اضافی خاصی موسوم به سرگره به ابتدای T اضافه می‌شود. هنگامی که این گرهٔ اضافی مورد استفاده قرار می‌گیرد، سه متغیر اشاره‌گر که آنها را HEAD (به جای ROOT) می‌نامیم و به سرگره اشاره می‌کند همچنین اشاره‌گر چپِ سرگره که به ریشهٔ T اشاره می‌کند. شکل ۱۸ـ ۷ تصویر درخت دودویی شکل ۱ـ ۷ را نشان می‌دهد که از نمایش پیوندی با یک سرگره استفاده می‌کند، آن را با شکل ۶ـ ۷مقایسه کنید.




شکل ۱۸ـ ۷

فرض کنید درخت دودویی T خالی است. آنگاه T همچنان حاوی یک سرگره است، اما اشاره‌گر چپ سرگره حاوی مقدار پوچ NULL است. بدین ترتیب شرط

LEFT[HEAD] = NULL

بیان می‌کند که درخت خالی است.

روش دیگر نمایش بالا برای یک درخت دودویی T ، از سرگره به عنوان نگهبان استفاده می‌کند. به بیان دیگر، اگر یک گره زیردرخت خالی داشته باشد، آنگاه فیلد اشاره‌گر زیر درخت حاوی آدرس سرگره به عوض مقدار پوچ NULL است. برطبق آن، هیچ اشاره‌گری آدرس غیرمجاز ندارد و شرط

**LEFT[HEAD] = HEAD**

بیان می‌کند که زیردرخت خالی است.

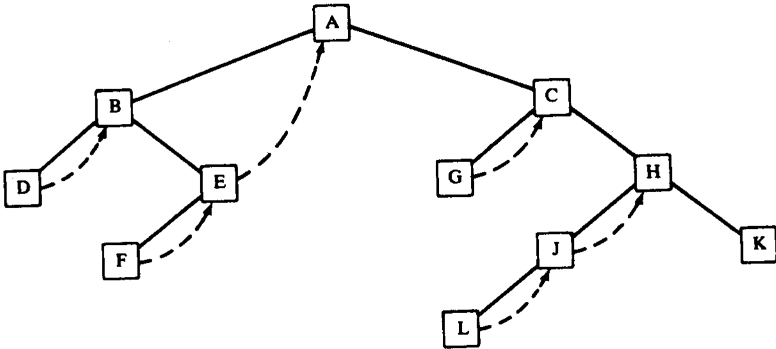
## نخ‌کشی‌ها، نخ‌کشیِ InOrder

بار دیگر نمایش پیوندی درخت دودویی T را درنظر بگیرید. تقریباً نصفِ ورودیهای فیلدهای اشاره‌گر LEFT و RIGHT شامل عناصر پوچ NULL است. این فضا با قراردادن نوع دیگری از اطلاعات به جای ورودیهای پوچ می‌تواند به شکل کاراتری مورد استفاده قرار گیرد. بطور مشخص ما اشاره‌گرهای خاصی را جانشین ورودیهای پوچ می‌کنیم که به گره‌های بالاترِ درخت اشاره می‌کند. این اشاره‌گرهای خاص را **نخ‌کشی‌ها** و درخت دودویی با این اشاره‌گرها را **درختهای نخ‌کشی شده** می‌گویند.

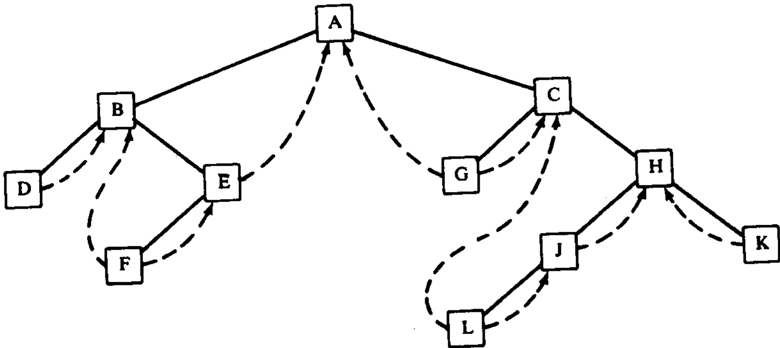
نخ‌کشی‌ها در یک درخت نخ‌کشی شده، باید به طریقی از اشاره‌گرهای معمولی تمیز داده شوند. در نمودار یک درخت نخ‌کشی شده، نخ‌کشی‌ها را معمولاً با خط‌چینها نمایش می‌دهند. در حافظه کامپیوتر یک فیلد TAG ، یک بیتی اضافی را می‌توان برای تمایز نخ‌کشی از اشاره‌گرهای معمولی، مورد استفاده قرار داد. یا به بیان دیگر، نخ‌کشی‌ها را وقتی اشاره‌گرهای معمولی با اعداد صحیح مثبت نمایش داده می‌شوند می‌توان با اعداد صحیح منفی نشان داد.

برای نخ‌کشی یک درخت دودویی راههای متعدد وجود دارد اما هر نخ‌کشی متناظر با یک پیمایش خاص T است. علاوه بر این می‌توان **نخ‌کشی یکطرفه** یا **نخ‌کشی دوطرفه** را انتخاب کرد. نخ‌کشی ما متناظر با پیمایش InOrder درخت T است مگر آن که خلاف آن بیان شود. بنابراین در نخ‌کشی یکطرفه T ، یک نخ‌کشی در فیلد RIGHT راستِ یک گره ظاهر می‌شود و به گرهٔ بعدی در پیمایش InOrder اشاره خواهد کرد و در نخ‌کشی دوطرفه T ، یک نخ‌کشی نیز در فیلد LEFT یک گره ظاهر می‌شود و به گرهٔ قبلی در پیمایش InOrder اشاره خواهد کرد. علاوه بر این وقتی T سرگره ندارد اشاره‌گر چپ گرهٔ اول و اشاره‌گر راست گرهٔ آخر (در پیمایش InOrder درخت T) حاوی مقدار پوچ NULL است، اما وقتی T یک سرگره دارد به سرگره اشاره می‌کند.

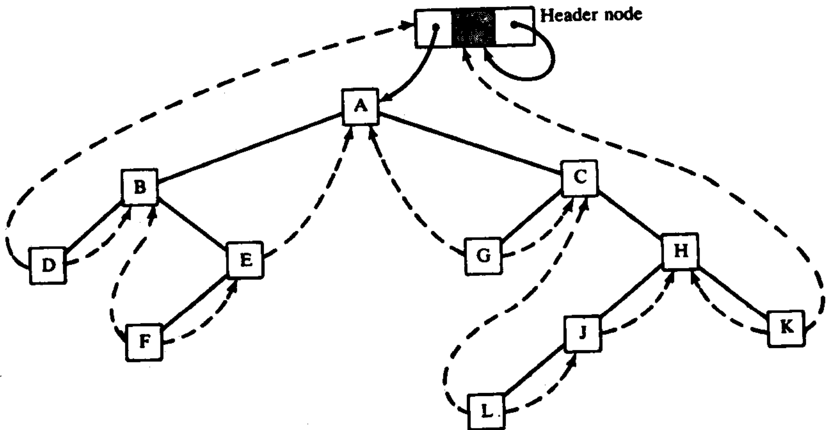
مشابه نخ‌کشی یکطرفه یک درخت دودویی T ، نخ‌کشی‌ای وجود دارد که متناظر با پیمایش PreOrder درخت T است. مسأله ۱۳-۷ را ببینید. ازطرف دیگر، هیچ نخ‌کشی‌ای وجود ندارد که متناظر با پیمایش PostOrder درخت T باشد.




(الف) نخ‌کشی InOrder یکطرفه




(ب) نخ‌کشی InOrder دوطرفه




(ج) نخ‌کشی دوطرفه با سرگره

شکل ۱۹ـ۷

## مثال ۱۲ـ ۷

درخت دودویی T شکل ۱ـ ۷ را درنظر بگیرید.

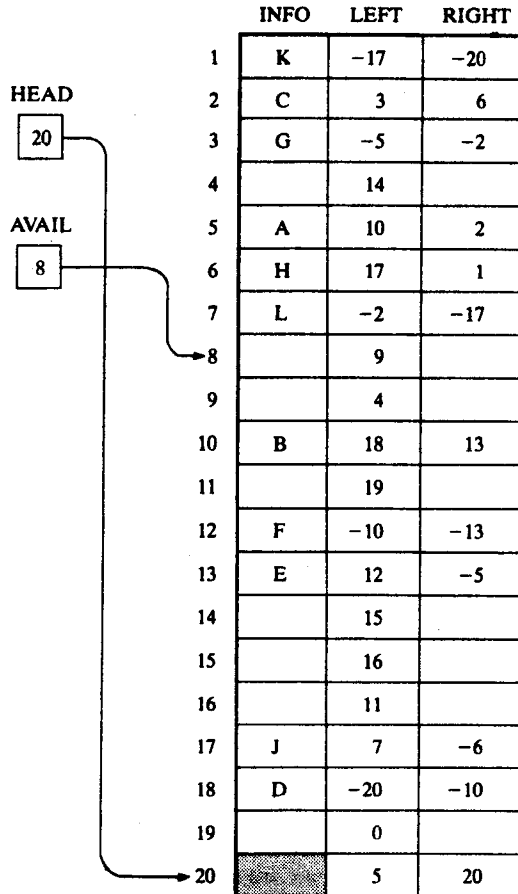
(الف) نخ‌کشی **InOrder** یکطرفه T در شکل ۱۹ـ ۷ (الف) رسم شده است. از آنجا که در پیمایش **InOrder** درخت T ، به A می‌توان پس از E دسترسی پیدا کرد، یک نخ‌کشی از گرهٔ E به گرهٔ A وجود دارد. ملاحظه می‌کنید یک نخ به‌استثنای گرهٔ K جانشین هر اشاره‌گر راست پوچ شده است که گرهٔ آخر در پیمایش **InOrder** درخت T است.

(ب) نخ‌کشی **Inorder** دوطرفه T در شکل ۱۹ـ ۷ (ب) رسم شده است. از آنجا که در پیمایش **InOder** درخت T ، به L می‌توان پس از C دسترسی پیدا کرد، یک نخ‌کشی چپ از گرهٔ L به گرهٔ C وجود دارد. ملاحظه می‌کنید که یک نخ به‌استثنای گرهٔ D جانشین هر اشاره‌گر چپ پوچ شده است که نخستین گرهٔ در پیمایش **InOrder** درخت T است. تمام نخ‌کشی‌های راست همانند شکل ۱۹ـ ۷ (الف) هستند.

(ج) نخ‌کشی **InOrder** دوطرفه درخت T وقتی T یک سرگره دارد در شکل ۱۹ـ ۷ (ج) رسم شده است. دراینجا نخ‌کشی چپ D و نخ‌کشی راست K به سرگره اشاره می‌کنند. نخ‌کشی‌های دیگر به همان صورتی است که در شکل ۱۹ـ ۷ (ب) ارائه شده است.

(د) شکل ۷ـ ۷ چگونگی نگهداری T را در حافظه با استفاده از یک نمایش پیوندی نشان می‌دهد. شکل ۲۰ـ ۷ نشان می‌دهد که چگونه این نمایش باید اصلاح شود تا T با استفاده از **INFO[20]** به عنوان یک سرگره یک درخت نخ‌کشی شده **InOrder** دوطرفه باشد.

ملاحظه می‌کنید که **LEFT[12] = − 10** ، به بیان دیگر یک نخ‌کشی چپ از گرهٔ F به گرهٔ B وجود دارد. بطور مشابه، **RIGHT[17] = − 6** به این معنی است که یک نخ‌کشی راست از گرهٔ J به گرهٔ H وجود دارد. بالاخره ملاحظه می‌کنید که **RIGHT[20] = 20** به عبارت دیگر یک اشاره‌گر راست معمولی از سرگره به خودش وجود دارد. اگر T خالی باشد، آنگاه قرار دهید **LEFT[20] = − 20** و به این معنی است که یک نخ‌کشی چپ از سرگره به خودش وجود دارد.




| | INFO | LEFT | RIGHT |
|---|---|---|---|
| 1 | K | −17 | −20 |
| 2 | C | 3 | 6 |
| 3 | G | −5 | −2 |
| 4 | | 14 | |
| 5 | A | 10 | 2 |
| 6 | H | 17 | 1 |
| 7 | L | −2 | −17 |
| 8 | | 9 | |
| 9 | | 4 | |
| 10 | B | 18 | 13 |
| 11 | | 19 | |
| 12 | F | −10 | −13 |
| 13 | E | 12 | −5 |
| 14 | | 15 | |
| 15 | | 16 | |
| 16 | | 11 | |
| 17 | J | 7 | −6 |
| 18 | D | −20 | −10 |
| 19 | | 0 | |
| 20 | | 5 | 20 |

شکل ۲۰ـ۷

## ۷ـ۷ درختهای جستجوی دودویی

این بخش یکی از مهم‌ترین ساختمان داده علم کامپیوتر یعنی یک **درخت جستجوی دودویی** را مورد بحث و بررسی قرار می‌دهد. این ساختمان به ما امکان می‌دهد تا یک عنصر را جستجو کنیم و آن را با زمان اجرای میانگین $f(n) = O(\log_2 n)$ پیدا کنیم. علاوه بر این به‌سادگی می‌توان عنصر را در این ساختمان داده اضافه کرد یا از آن حذف کرد. این ساختمان داده در مقابل ساختمان‌های زیر قرار دارد:

**(الف) آرایهٔ مرتب‌شدهٔ خطی.** در اینجا می‌توان یک عنصر را جستجو کرد و آن را با زمان اجرای میانگین

$f(n) = O(\log_2 n)$ پیدا کرد اما اضافه‌کردن و حذف عنصر پرهزینه است.

**(ب) لیست پیوندی.** در اینجا به‌سادگی می‌توان عناصر را اضافه یا حذف کرد، اما در این روش جستجوی عنصر و پیداکردن آن پرهزینه است، چون باید از جستجوی **خطی** با زمان اجرای $f(n) = O(n)$ استفاده کرد.

هرچند هر گره در یک درخت جستجوی دودویی می‌تواند شامل تمام رکورد داده‌ها باشد اما تعریف درخت دودویی بستگی به فیلد داده شده دارد که مقادیر آن متمایز هستند و می‌تواند مرتب شده باشد.

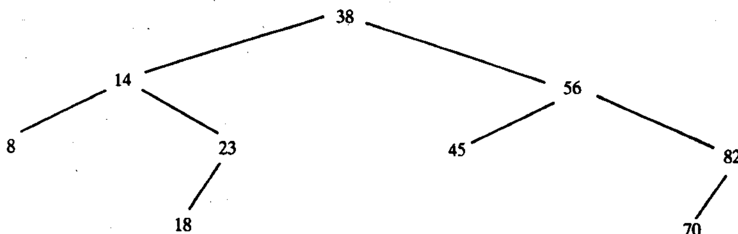
فرض کنید T یک درخت دودویی باشد. آنگاه T یک **درخت جستجوی دودویی** (یا **درخت مرتب‌شدهٔ دودویی**) نامیده می‌شود اگر هر گرهٔ N درخت T دارای خاصیت زیر باشد:

*مقدار* در N *بزرگتر از* هر *مقدار* در زیردرخت چپ N و *کوچکتر از* هر *مقدار* در زیردرخت *راست* N *است.*

به‌آسانی ملاحظه می‌شود که این خاصیت تضمین می‌کند که پیمایش InOrder درخت T باعث می‌شود لیست عناصر T مرتب‌شده باشند.

## مثال ۱۳ـ ۷

درخت دودویی T شکل ۲۱ـ ۷ را درنظر بگیرید.




**شکل ۲۱ـ ۷**

T یک درخت جستجوی دودویی است، یعنی هر گرهٔ N در T از هر عدد زیردرخت چپ آن بزرگتر و از هر عدد زیردرخت راست آن کوچکتر است. فرض کنید عدد 35 جانشین عدد 23 شده است آنگاه T همچنان یک درخت جستجوی دودویی خواهد بود. ازطرف دیگر اگر عدد 40 را جایگزین عدد 23 کنیم T یک درخت جستجوی دودویی نخواهد بود، چون در زیردرخت چپ عدد 38 بزرگتر از 40 نیست.

**(ب)** فایل شکل ۸ـ ۷ را درنظر بگیرید. همانگونه که در شکل ۹ـ ۷ گفته شد، این فایل نسبت به کلید NAME یک درخت جستجوی دودویی است. ازطرف دیگر این فایل نسبت به کلید شمارهٔ تأمین اجتماعی SSN یک درخت جستجوی دودویی نیست. این وضعیت مشابه یک آرایه از رکوردها است که

نسبت به یک کلید مرتب شده اما نسبت به هیچ کلید دیگر مرتب شده نباشد.

تعریف یک درخت جستجوی دودویی داده شده در این بخش براین فرض استوار است که مقادیر تمام گره‌ها متمایز هستند. تعریف مشابه‌ای از یک درخت جستجوی دودویی وجود دارد که اجازهٔ مساوی بودن دو مقدار را به ما می‌دهد به بیان دیگر، هر گره N در آن دارای خاصیت زیر است :

*مقدار در N بزرگتر از هر مقدار در زیردرخت چپ N و کوچکتر یا مساوی هر مقدار در زیردرخت راست N است.* هنگام استفاده از این تعریف، عملیات بخش بعد باید براساس آن تغییر کند.

## ۸ـ ۷ جستجو و واردکردن یک عنصر در درختهای جستجوی دودویی

فرض کنید T یک درخت جستجوی دودویی باشد. این بخش عملیات اصلی و پایه جستجو و واردکردن یک عنصر را در T توضیح می‌دهد. درواقع، جستجوکردن و واردکردن یک عنصر، تنها با یک الگوریتم جستجو و واردکردن انجام می‌شود. عمل حذف عنصر در بخش بعد بررسی می‌شود. پیمایش درخت T دقیقاً مانند پیمایش هر درخت دودویی است، این مبحث در بخش ۴ـ ۷ گنجانده شده است.

فرض کنید عنصر اطلاعاتی ITEM داده شده است. الگوریتم زیر مکان ITEM را در درخت جستجوی دودویی پیدا می‌کند یا ITEM را به عنوان یک گرهٔ جدید در مکان مربوطه‌اش در درخت اضافه می‌کند.

**(الف)** ITEM را با N گرهٔ ریشهٔ درخت مقایسه کنید.

**(i)** اگر ITEM<N ، به طرف بچهٔ چپ N پیش بروید.

**(ii)** اگر ITEM>N ، به طرف بچهٔ راست N پیش بروید.

**(ب)** مرحلهٔ (الف) را تکرار کنید تا یکی از حالتهای زیر اتفاق بیفتد:

**(i)** گرهٔ N را وقتی ITEM = N است ملاقات کنید. در این حالت جستجو موفق است.

**(ii)** یک زیردرخت خالی را ملاقات کنید که بیان می‌کند جستجو موفق نیست و ITEM را به جای زیردرخت خالی اضافه کنید.

به بیان دیگر، از ریشهٔ R در درخت T به طرف پائین پیش بروید تا ITEM در T پیدا شود یا ITEM را به عنوان گرهٔ انتهایی در T اضافه کنید.

### مثال ۱۴ـ ۷

**(الف)** درخت جستجوی دودویی T شکل ۲۱ـ ۷ را در نظر بگیرید. فرض کنید ITEM = 20 داده شده است. با شبیه‌سازی الگوریتم بالا به مرحله‌های زیر می‌رسیم :

۱ـ ITEM = 20 را با ریشهٔ درخت T یعنی 38 مقایسه کنید. چون 20<38 به طرف بچهٔ چپ 38 که 14

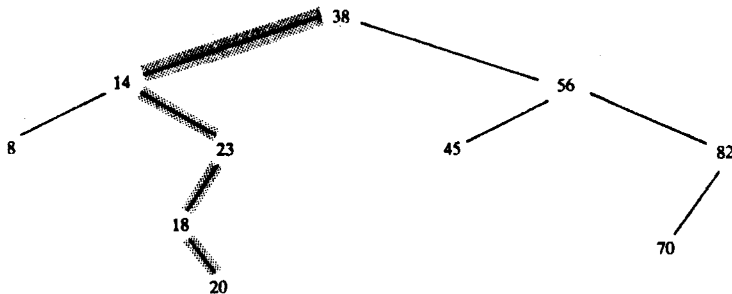
است پیش بروید.

۲ـ ITEM = 20 را با 14 مقایسه کنید. چون 14<20 به طرف بچهٔ راست 14 که 23 است پیش بروید.

۳ـ ITEM = 20 را با 23 مقایسه کنید. چون 23>20 به طرف بچهٔ چپ 23 که 18 است پیش بروید.

۴ـ ITEM = 20 را با 18 مقایسه کنید. چون 18<20 و 18 بچهٔ راست ندارد. 20 را به عنوان بچهٔ راست 18 اضافه کنید.

شکل ۲۲ـ ۷ درخت جدیدی را نشان می‌دهد که عنصر ITEM = 20 به آن اضافه شده است.




**شکل ۲۲ـ ۷** ITEM = 20 اضافه شده است.

یالهای هاشورخورده و سایه‌دار، بیان می‌کند که این مسیر هنگام اجرای الگوریتم تا پائین طی می‌شود.

(ب) درخت جستجوی دودویی T شکل ۹ـ ۷ را در نظر بگیرید. فرض کنید ITEM = Davis داده شده است. با شبیه‌سازی الگوریتم بالا، مراحل زیر حاصل می‌شود:

۱ـ ITEM = Davis را با ریشهٔ درخت، Harris مقایسه کنید. چون Davis<Harris ، به طرف بچهٔ چپ Harris که Cohen است پیش بروید.

۲ـ ITEM = Davis را با Cohen مقایسه کنید. چون Davis>Cohen ، به طرف بچهٔ راست Cohen که Green است پیش بروید.

۳ـ ITEM = Davis را با Green مقایسه کنید. چون Davis<Green ، به طرف بچهٔ چپ Green که در Davis است پیش بروید.

۴ـ ITEM = Davis را با بچهٔ چپ Davis مقایسه کنید. از این‌رو مکان Davis در درخت پیدا شده است.

## مثال ۱۵ـ ۷

فرض کنید شش عدد زیر به‌ترتیب در یک درخت جستجوی دودویی خالی اضافه شده است :

**40, 60, 50, 33, 55, 11**

شکل ۲۳ـ ۷ شش مرحله از درخت را نشان می‌دهد. تأکید می‌کنیم که اگر شش عدد داده شده با ترتیب مختلف داده شده باشد، آنگاه درخت‌ها ممکن است با هم فرق کنند و عمق مختلف داشته باشند. نمایش رسمی الگوریتم جستجو و اضافه‌کردن از زیربرنامهٔ Procedure زیر استفاده می‌کند، که مکان یک ITEM داده شده و پدر آن را پیدا می‌کند. زیربرنامهٔ با استفاده از اشاره‌گر PTR و اشاره‌گر SAVE برای گره پدر به طرف پائین درخت را پیمایش می‌کند. این زیربرنامه در بخش بعد هنگام حذف عنصرها مورد استفاده قرار خواهد گرفت :

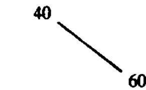
**Procedure 7.4:** FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR)
A binary search tree T is in memory and an ITEM of information is given. This procedure finds the location LOC of ITEM in T and also the location PAR of the parent of ITEM. There are three special cases:
(i) LOC = NULL and PAR = NULL will indicate that the tree is empty.
(ii) LOC ≠ NULL and PAR = NULL will indicate that ITEM is the root of T.
(iii) LOC = NULL and PAR ≠ NULL will indicate that ITEM is not in T and can be added to T as a child of the node N with location PAR.

1. [Tree empty?]
   If ROOT = NULL, then: Set LOC := NULL and PAR := NULL, and Return.
2. [ITEM at root?]
   If ITEM = INFO[ROOT], then: Set LOC := ROOT and PAR := NULL, and Return.
3. [Initialize pointers PTR and SAVE.]
   If ITEM < INFO[ROOT], then:
       Set PTR := LEFT[ROOT] and SAVE := ROOT.
   Else:
       Set PTR := RIGHT[ROOT] and SAVE := ROOT.
   [End of If structure.]
4. Repeat Steps 5 and 6 while PTR ≠ NULL:
5.     [ITEM found?]
       If ITEM = INFO[PTR], then: Set LOC := PTR and PAR := SAVE, and Return.
6.     If ITEM < INFO[PTR], then:
           Set SAVE := PTR and PTR := LEFT[PTR].
       Else:
           Set SAVE := PTR and PTR := RIGHT[PTR].
       [End of If structure.]
   [End of Step 4 loop.]
7. [Search unsuccessful.] Set LOC := NULL and PAR := SAVE.
8. Exit.

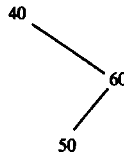
در مرحلهٔ 6 ملاحظه می‌کنید که بسته به این که **ITEM<INFO[PTR]** یا **ITEM>INFO[PTR]** به طرف بچهٔ چپ یا بچهٔ راست حرکت می‌کنیم.

40

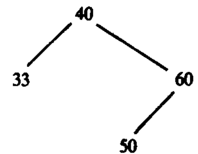
(1) ITEM = 40.



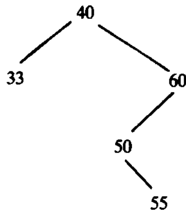

(2) ITEM = 60.



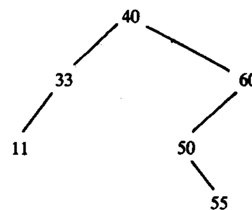

(3) ITEM = 50.




(4) ITEM = 33.




(5) ITEM = 55.




(6) ITEM = 11.

شکل ۲۳ـ۷

بیان رسمی الگوریتم جستجو و اضافه‌کردن به شرح زیر است :

**Algorithm 7.5:** INSBST(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM, LOC)
A binary search tree T is in memory and an ITEM of information is given. This algorithm finds the location LOC of ITEM in T or adds ITEM as a new node in T at location LOC.

1. Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR). [Procedure 7.4.]
2. If LOC ≠ NULL, then Exit.
3. [Copy ITEM into new node in AVAIL list.]
   (*a*) If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
   (*b*) Set NEW := AVAIL, AVAIL := LEFT[AVAIL] and INFO[NEW] := ITEM.
   (*c*) Set LOC := NEW, LEFT[NEW] := NULL and RIGHT[NEW] := NULL.
4. [Add ITEM to tree.]
   If PAR = NULL, then:
       Set ROOT := NEW.
   Else if ITEM < INFO[PAR], then:
       Set LEFT[PAR] := NEW.
   Else:
       Set RIGHT[PAR] := NEW.
   [End of If structure.]
5. Exit.

در مرحلهٔ 4 ملاحظه می‌کنید که سه حالت ممکن وجود دارد: (۱) درخت خالی باشد (۲) ITEM به عنوان یک بچهٔ چپ اضافه شود و (۳) ITEM به عنوان یک بچهٔ راست اضافه شود.

## پیچیدگی الگوریتم جستجوی عنصر

فرض کنید در یک درخت جستجوی دودویی T می‌خواهیم یک عنصر اطلاعاتی را جستجو کنیم. ملاحظه می‌کنید که تعداد مقایسه‌ها محدود به عمق درخت می‌شود. این موضوع از این واقعیت ناشی می‌شود که ما از یک مسیر درخت به طرف پائین پیش می‌رویم. بنابراین، زمان اجرای جستجو متناسب با عمق درخت است.

فرض کنید n عنصر اطلاعاتی $A_1, A_2, ..., A_N$ داده شده است و فرض کنید عناصر به ترتیب در یک درخت جستجوی دودویی T اضافه می‌شوند. یادآوری می‌کنیم که برای n عنصر تعداد !n جـایگشت وجود دارد (بخش ۲ـ ۲). هر یک از چنین جایگشتی باعث به‌وجود آمـدنِ درخت مـربوط بـه خـود می‌شود. می‌توان نشان داد که عمق میانگین !n درخت تقریباً برابر $c \log_2 n$ است کـه در آن $c = 1.4$ . بنابراین، زمان اجرای میانگین $f(n)$ جستجو یک عنصر در درخت دودویی T با n عنصر مـتناسب بـا $\log_2 n$ است یعنی $f(n) = O(\log_2 n)$ .

## کاربرد درختهای جستجوی دودویی

مجموعه‌ای از n عنصر اطلاعاتی $A_1, A_2, ..., A_N$ را درنظر بگیرید. فرض کنید بخواهیم تمام عناصر تکراری را که در این مجموعه وجود دارند پیدا کرده آنها را حذف کنیم. یک راه ساده برای این منظور به شرح زیر است :

**الگوریتم A**: عناصر را از $A_1$ تا $A_N$ یعنی از چپ به راست بخوانید.

**(الف)** برای هر عنصر $A_K$ ، $A_K$ را با $A_1, A_2, ..., A_{K-1}$ مقایسه کنید یعنی $A_K$ را با عناصری که قبل از $A_k$ هستند مقایسه کنید.

**(ب)** اگر $A_k$ در بین $A_1, A_2, ..., A_{K-1}$ وجود داشت، آنگاه $A_K$ را حذف کنید.

پس از آن که تمام عناصر خوانده شده و مورد بررسی قرار گرفت، آنگاه در این مـجموعه عـناصر تکراری نخواهد بود.

### مثال ۱۶ـ ۷

فرض کنید الگوریتم A بر لیست ۱۵ عددی زیر بکار گرفته شد :

**14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23**

ملاحظه می‌کنید که چهار عدد اول یعنی 14، 10، 17 و 12 حذف نمی‌شوند. بنابراین :

$A_5 = 10$ حذف می‌شود چون $A_5 = A_2$

$A_8 = 12$ حذف می‌شود چون $A_8 = A_4$

$A_{11} = 20$ حذف می‌شود چون $A_{11} = A_7$

$A_{14} = 11$ حذف می‌شود چون $A_{14} = A_6$

هنگامی که اجرای الگوریتم A به پایان می‌رسد، ۱۱ عدد

14, 10, 17, 12, 11, 20, 18, 25, 8, 22, 23

که همگی متمایز هستند باقی می‌ماند.

حال پیچیدگی زمانی الگوریتم A را درنظر بگیرید که به‌وسیلهٔ تعداد مقایسه‌ها تعیین می‌شود. قبل از همه، فرض می‌کنیم که d تعداد عناصر دوتایی تکراری در مقایسه با n تعداد عناصر اطلاعاتی بسیار کوچک است. ملاحظه می‌کنید مرحله‌ای که شامل $A_K$ است به‌طور تقریبی به $K-1$ مقایسه احتیاج دارد چون $A_k$ با عنصرهای $A_1, A_2, \ldots, A_{K-1}$ (کمتر ممکن است تابحال حذف شده باشند) مقایسه می‌شود. بنابراین، $f(n)$ تعداد مقایسه‌ها موردنیاز در الگوریتم A تقریباً برابر

$$0 + 1 + 2 + 3 + \ldots + (n-2) + (n-1) = \frac{(n-1)n}{2} = O(n^2)$$

است. برای مثال، برای $n = 1000$ عنصر، الگوریتم A تقریباً به 500 000 مقایسه احتیاج دارد. به بیان دیگر، زمان اجرای الگوریتم A متناسب با $n^2$ است.

با استفاده از یک درخت جستجوی دودویی می‌توان الگوریتم دیگر نوشت که عناصر دوتایی تکراری را از یک مجموعهٔ n عنصری $A_1, A_2, \ldots, A_N$ پیدا کنند.

**الگوریتم B** : با استفاده از عناصر $A_1, A_2, \ldots, A_N$ یک درخت جستجوی دودویی بسازید. هنگام ساختن درخت، در صورتی که مقدار $A_k$ قبلاً در درخت ظاهر شده باشد $A_k$ را از لیست حذف کنید.

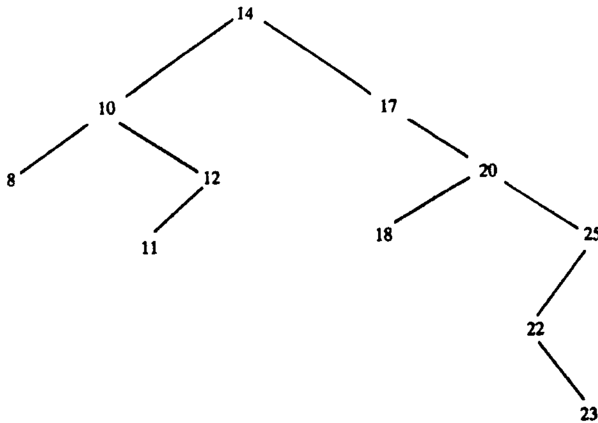
مزیت اصلی الگوریتم B آن است که هر عنصر $A_k$ تنها با عنصرهای یک شاخهٔ درخت مقایسه می‌شود. می‌توان نشان داد که طول میانگین چنین شاخه‌ای تقریباً برابر $c \log_2 K$ است که‌در آن $c = 1.4$ . بنابراین $f(n)$ تعداد کل مقایسه‌های موردنیاز در الگوریتم B تقریباً به $n \log_2 n$ مقایسه نیاز دارد. برای مثال، برای $n = 1000$ ، الگوریتم B مستلزم تقریباً 10000 مقایسه است که در الگوریتم A ، تعداد مقایسه‌ها برابر 500000 است. متذکر می‌شویم که در بدترین حالت، تعداد مقایسه‌های الگوریتم B ، برابر تعداد مقایسه‌های الگوریتم A است.

## مثال ۱۷ـ۷

مجدداً لیست ۱۵ عددی زیر را درنظر بگیرید :

14, 10, 17, 12, 10, 11, 20, 12, 18, 25, 20, 8, 22, 11, 23

با اعمال الگوریتم B بر این لیست عددی، درخت شکل ۲۴ـ۷ به دست می‌آید :




شکل ۲۴ـ۷

تعداد دقیق مقایسه‌ها برابر است با:

$$0 + 1 + 1 + 2 + 2 + 3 + 2 + 3 + 3 + 3 + 3 + 2 + 4 + 4 + 5 = 38$$

از طرف دیگر الگوریتم A نیازمند

$$0 + 1 + 2 + 3 + 2 + 4 + 5 + 4 + 6 + 7 + 6 + 8 + 9 + 5 + 10 = 27$$

مقایسه است.

## ۹ـ۷ حذف یک عنصر از یک درخت جستجوی دودویی

فرض کنید T یک درخت جستجوی دودویی است و عنصر اطلاعاتی ITEM داده شده است. این بخش الگوریتمی را ارائه می‌دهد که عنصر ITEM را از درخت T حذف می‌کند.

الگوریتم حذف در وهلهٔ اول از زیربرنامهٔ Procedure 7.4 استفاده می‌کند تا مکان گرهٔ N را که حاوی عنصر ITEM است و همچنین مکان گرهٔ پدر $P(N)$ را پیدا می‌کند. روشی که با آن N از درخت حذف می‌شود در درجهٔ اول بستگی به تعداد بچه‌های N دارد. سه حالت وجود دارد:

حالت ۱ : N بچه‌ای ندارد. آنگاه تنها با جایگزین‌شدن مکان N در گرهٔ پدر $P(N)$ به وسیلهٔ اشاره‌گر پوچ null گرهٔ N از درخت حذف می‌شود.

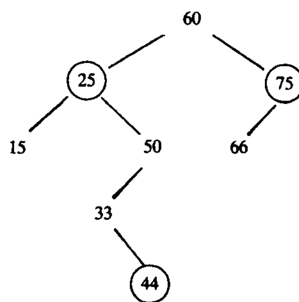
حالت ۲ : N دقیقاً یک بچه دارد. آنگاه تنها با جایگزین‌شدن مکان N در $P(N)$ به وسیلهٔ مکان تنهابچهٔ N ، گرهٔ N از درخت حذف می‌شود.

**حالت ۳:** N دو بچه دارد. فرض کنید S(*N*) نمایش ریشهٔ بعدی پیمایش **InOrder**، گرهٔ N باشد. دانشجو می‌تواند تحقیق کند که S(*N*) بچهٔ چپ ندارد. آنگاه نخست با حذف S(*N*) از **T** (با استفاده از حالت ۱ یا حالت ۲) و سپس با جانشین‌کردن گرهٔ S(*N*) به جای گرهٔ N در درخت T، گرهٔ N از **T** حذف می‌شود.

ملاحظه می‌کنید که حالت سوم پیچیده‌تر از دو حالت اول است. در تمام سه حالت بالا، فضای حافظهٔ گره حذف شدهٔ N به لیست **AVAIL** برگردانده می‌شود.

## مثال ۱۸ـ۷

درخت جستجوی دودویی شکل ۲۵ـ۷ (الف) را درنظر بگیرید. فرض‌کنید **T** به‌صورت شکل ۲۵ـ۷ (ب) در حافظه قرار می‌گیرد.




(الف) قبل از حذف

ROOT = 3

AVAIL = 5

| | INFO | LEFT | RIGHT |
|---|---|---|---|
| 1 | 33 | 0 | 9 |
| 2 | 25 | 8 | 10 |
| 3 | 60 | 2 | 7 |
| 4 | 66 | 0 | 0 |
| 5 | | 6 | |
| 6 | | 0 | |
| 7 | 75 | 4 | 0 |
| 8 | 15 | 0 | 0 |
| 9 | 44 | 0 | 0 |
| 10 | 50 | 1 | 0 |

(ب) نمایش پیوندی

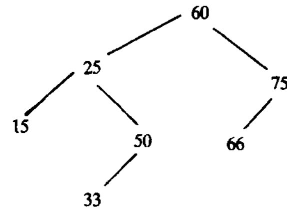
شکل ۲۵ـ۷

**(الف)** فرض کنید گرهٔ 44 از درخت **T** شکل ۲۵ـ۷ حذف شده است. توجه دارید که گرهٔ 44 بچه‌ای ندارد. شکل ۲۶ـ۷ (الف) این درخت را پس از حذف 44 نشان می‌دهد و شکل ۲۶ـ۷ (ب) نمایش پیوندی آن را در حافظه نشان می‌دهد.

| | INFO | LEFT | RIGHT |
|---|---|---|---|
| 1 | 33 | 0 | 0 |
| 2 | 25 | 8 | 10 |
| 3 | 60 | 2 | 7 |
| 4 | 66 | 0 | 0 |
| 5 | | 6 | |
| 6 | | 0 | |
| 7 | 75 | 4 | 0 |
| 8 | 15 | 0 | 0 |
| 9 | | 5 | |
| 10 | 50 | 1 | 0 |

ROOT: 3

AVAIL: 9




(الف) گرهٔ 44 حذف می‌شود.

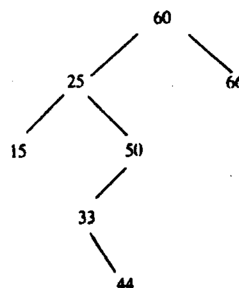
(ب) نمایش پیوندی

شکل ۲۶ـ ۷

عمل حذف تنها با جایگزینی NULL در گرهٔ پدر یعنی 33 انجام شده است. مکانهای هاشورخورده بیانگر این تغییرات است.

**(ب)** فرض کنید به جای گرهٔ 44، گرهٔ 75 از درخت T شکل ۲۵ـ ۷ حذف می‌شود. توجه دارید که گرهٔ 75 تنها یک بچه دارد. شکل ۲۷ـ ۷ (الف) این درخت را پس از حذف گرهٔ 75 نشان می‌دهد و شکل ۲۷ ـ ۷ (ب) نمایش پیوندی آن را نشان می‌دهد. عمل حذف تنها با تغییر اشاره‌گر راست گرهٔ پدر 60 انجام می‌شود که در آغاز به 75 اشاره می‌کرد، درحالی که اکنون به گرهٔ 66 اشاره می‌کند که تنها بچهٔ گرهٔ 75 است. مکانهای هاشورخورده بیانگر این تغییرات است.

**(ج)** فرض کنید به جای گرهٔ 44 یا گرهٔ 75، گرهٔ 25 از درخت T شکل ۲۵ـ ۷ حذف می‌شود. توجه دارید که گرهٔ 25 دو بچه دارد. علاوه بر این ملاحظه می‌کنید که گرهٔ 33 ریشهٔ بعدی پیمایش InOrder گرهٔ 25 است. شکل ۲۸ـ ۷ (الف) این درخت را پس از حذف 25 نشان می‌دهد و شکل ۲۸ـ ۷ (ب) نمایش پیوندی آن را نشان می‌دهد. عمل حذف نخست با حذف 33 از درخت و بدنبال آن با جانشینی گرهٔ 33 به جای گرهٔ 25 انجام می‌شود. تأکید می‌کنم که جانشینی گرهٔ 33 به جای 25، در حافظه تنها با تغییر اشاره‌گرها انجام می‌شود نه با جابجایی محتوای یک گره از یک مکان به مکان دیگر. بدین ترتیب 33 همچنان مقدار INFO[1] است.




(الف) گرهٔ 75 حذف می‌شود.

ROOT: 3

AVAIL: 7

| | INFO | LEFT | RIGHT |
|---|---|---|---|
| 1 | 33 | 0 | 9 |
| 2 | 25 | 8 | 10 |
| 3 | 60 | 2 | 4 |
| 4 | 66 | 0 | 0 |
| 5 | | 6 | |
| 6 | | 0 | |
| 7 | | [illegible] | |
| 8 | 15 | 0 | 0 |
| 9 | 44 | 0 | 0 |
| 10 | 50 | 1 | 0 |

(ب) نمایش پیوندی

شکل ۲۷ـ ۷

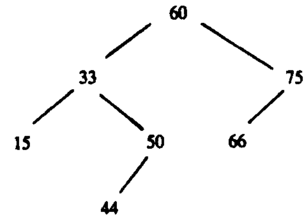
الگوریتم حذف ما، برحسب زیربرنامه‌های Procedure 7.6 و Procedure 7.7 بـه شـرح زیـر بـیان می‌شود. اولین زیربرنامه به حالتهای ۱ و ۲ مربوط می‌شود که در آن گرهٔ حذف‌شدهٔ N دو بچه ندارد و زیربرنامهٔ دوم به حالت ۳ مربوط می‌شود که در آن N دو بچه دارد. حالتهای کوچک متعددی وجود دارد که منعکس‌کنندهٔ این واقعیت است که N می‌تواند "بچهٔ چپ" بچهٔ راست یا ریشه باشد. هـمچنین در حالت ۲، N می‌تواند یک بچهٔ چپ یا یک بچهٔ راست داشته باشد.

زیربرنامهٔ Procedure 7.7 حالتی را مورد بررسی قرار می‌دهد که گـرهٔ حـذف‌شدهٔ N دو بـچه دارد. متذکر می‌شویم که ریشهٔ بعدی پیمایش InOrder را می‌توان با جابجائی بچهٔ راست N بدست آورد و آنگاه به طور مکرر به طرف چپ جابجا می‌کنیم تا گره‌ای با زیردرخت چپ خالی ملاقات شود.

| | INFO | LEFT | RIGHT |
|---|---|---|---|
| 1 | 33 | [illegible] | 10 |
| 2 | | | |
| 3 | 60 | [illegible] | 7 |
| 4 | 66 | 0 | 0 |
| 5 | | 6 | |
| 6 | | 0 | |
| 7 | 75 | 4 | 0 |
| 8 | 15 | 0 | 0 |
| 9 | 44 | 0 | 0 |
| 10 | 50 | [illegible] | 0 |

ROOT: 3

AVAIL: [illegible]




(الف) گرهٔ 25 حذف می‌شود.

(ب) نمایش پیوندی

شکل ۲۸ـ۷

**Procedure 7.6:** CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure deletes the node N at location LOC, where N does not have two children. The pointer PAR gives the location of the parent of N, or else PAR = NULL indicates that N is the root node. The pointer CHILD gives the location of the only child of N, or else CHILD = NULL indicates N has no children.

1. [Initializes CHILD.]
   If LEFT[LOC] = NULL and RIGHT[LOC] = NULL, then:
       Set CHILD := NULL.
   Else if LEFT[LOC] ≠ NULL, then:
       Set CHILD := LEFT[LOC].
   Else
       Set CHILD := RIGHT[LOC].
   [End of If structure.]
2. If PAR ≠ NULL, then:
       If LOC = LEFT[PAR], then:
           Set LEFT[PAR] := CHILD.
       Else:
           Set RIGHT[PAR] := CHILD.
       [End of If structure.]
   Else:
       Set ROOT := CHILD.
   [End of If structure.]
3. Return.

**Procedure 7.7:** CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR)

This procedure will delete the node N at location LOC, where N has two children. The pointer PAR gives the location of the parent of N, or else PAR = NULL indicates that N is the root node. The pointer SUC gives the location of the inorder successor of N, and PARSUC gives the location of the parent of the inorder successor.

```
1. [Find SUC and PARSUC.]
   (a) Set PTR := RIGHT[LOC] and SAVE := LOC.
   (b) Repeat while LEFT[PTR] ≠ NULL:
           Set SAVE := PTR and PTR := LEFT[PTR].
       [End of loop.]
   (c) Set SUC := PTR and PARSUC := SAVE.
2. [Delete inorder successor, using Procedure 7.6.]
   Call CASEA(INFO, LEFT, RIGHT, ROOT, SUC, PARSUC).
3. [Replace node N by its inorder successor.]
   (a) If PAR ≠ NULL, then:
           If LOC = LEFT[PAR], then:
               Set LEFT[PAR] := SUC.
           Else:
               Set RIGHT[PAR] := SUC.
           [End of If structure.]
       Else:
           Set ROOT := SUC.
       [End of If structure.]
   (b) Set LEFT[SUC] := LEFT[LOC] and
       RIGHT[SUC] := RIGHT[LOC].
4. Return.
```

اکنون می‌توانیم با استفاده از زیربرنامه‌های پایه و اصلیِ 7.6 و 7.7 به عنوان آجرهای ساختمانی و سنگ بنا، الگوریتم حذف را به صورت رسمی بیان کنیم:

**Algorithm 7.8:** DEL(INFO, LEFT, RIGHT, ROOT, AVAIL, ITEM)

A binary search tree T is in memory, and an ITEM of information is given. This algorithm deletes ITEM from the tree.

```
1. [Find the locations of ITEM and its parent, using Procedure 7.4.]
   Call FIND(INFO, LEFT, RIGHT, ROOT, ITEM, LOC, PAR).
2. [ITEM in tree?]
   If LOC = NULL, then: Write: ITEM not in tree, and Exit.
3. [Delete node containing ITEM.]
   If RIGHT[LOC] ≠ NULL and LEFT[LOC] ≠ NULL, then:
       Call CASEB(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
   Else:
       Call CASEA(INFO, LEFT, RIGHT, ROOT, LOC, PAR).
   [End of If structure.]
4. [Return deleted node to the AVAIL list.]
   Set LEFT[LOC] := AVAIL and AVAIL := LOC.
5. Exit.
```

## ۷ـ۱۰ HeapSort ، Heap

این بخش ساختمان درخت دیگری را مورد بحث و بررسی قرار می‌دهد که **Heap** نامیده می‌شود. **Heap** در یک الگوریتمِ مرتب‌کردن جالب و زیبا موسوم به **HeapSort** بکار برده می‌شود. اگرچه روشهای مرتب‌کردن به‌طور اساسی در فصل ۹ بررسی می‌شود اما در این بخش الگوریتم **HeapSort** ارائه می‌شود و پیچیدگی آن با الگوریتم مرتب‌کردن حبابی و الگوریتم **QuickSort** که به ترتیب در فصل‌های ۴ و ۶ توضیح داده شدند مقایسه می‌شود.

فرض کنید **H** یک درخت دودویی کامل با **n** عنصر باشد. فرض می‌کنیم که **H** در حافظه به وسیلهٔ آرایهٔ خطی **TREE** و با استفاده از نمایش ترتیبی **H** نگهداری می‌شود نه با نمایش پیوندی، مگر آن که خلاف آن بیان شود آنگاه **H** یک **Heap** یا یک **MaxHeap** نامیده می‌شود مشروط بر این که هر گرهٔ **N** از **H** دارای خاصیت زیر باشد :

مقدار در **N** بزرگتر یا مساوی با مقدار در هر بچهٔ **N** است. بنابراین مقدار در **N** بزرگتر یا مساوی با مقدار در هر نسل **N** است. یک **MinHeap** به صورت مشابه تعریف می‌شود: مقدار در **N** کوچکتر یا مساوی با مقدار در هر بچهٔ **N** است.

### مثال ۱۹ـ۷

درخت کامل **H** شکل ۲۹ـ ۷ (الف) را درنظر بگیرید. ملاحظه می‌کنید که **H** یک **Heap** است. در حالت خاص،بزرگترین عنصر در **H** در "بالای" **Heap** یعنی در ریشهٔ درخت است. شکل ۲۹ـ ۷ (ب) نمایش ترتیبی **H** را به وسیلهٔ آرایه **TREE** نشان می‌دهد. به بیان دیگر، **TREE[1]** ریشهٔ درخت **H** همچنین بچه چپ و راست گرهٔ **TREE[K]** به ترتیب **TREE[2K]** و **TREE[2K + 1]** هستند. در حالت خاص پدر هر گرهٔ غیر ریشه **TREE[J]** گرهٔ **TREE[J ÷ 2]** است که در آن منظور از **J ÷ 2** تقسیم صحیح است. ملاحظه می‌کنید که گره‌های هم‌سطح در **H** یکی پس از دیگری در آرایهٔ **TREE** ظاهر می‌شود.

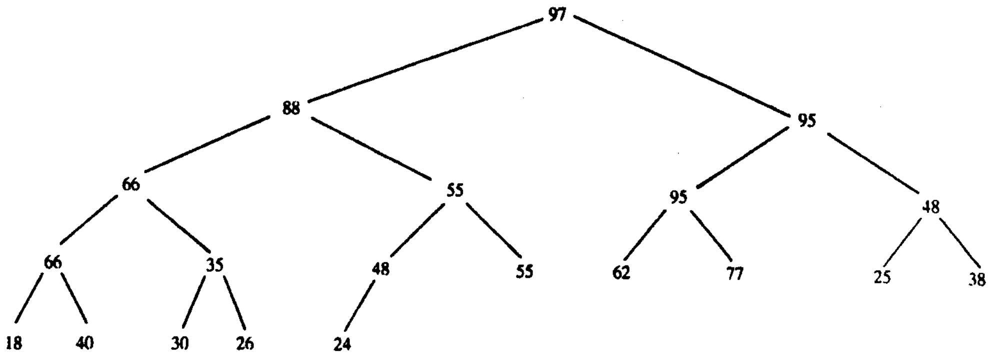
### اضافه‌کردن یک عنصر در Heap

فرض کنید **H** یک **Heap** با **N** عنصر باشد همچنین فرض کنید یک عنصر اطلاعاتی **ITEM** داده شده است. عنصر **ITEM** را به داخل **Heap** به صورت زیر اضافه می‌کنیم :

(۱) نخست عنصر **ITEM** را به انتهای **H** طوری اضافه می‌کنیم که **H** همچنان یک درخت کامل باشد، اما الزاماً یک **Heap** نیست.

(۲) آنگاه اجازه دهید **ITEM** به جای مربوطه‌اش در **H** طوری بالا رود که **H** نهایتاً یک **Heap** باشد.

قبل از آن که زیربرنامه **Procedure** را بیان کنیم چگونگی کار این زیربرنامه را توضیح می‌دهیم.




**Heap (الف)**

**TREE**

| 97 | 88 | 95 | 66 | 55 | 95 | 48 | 66 | 35 | 48 | 55 | 62 | 77 | 25 | 38 | 18 | 40 | 30 | 26 | 24 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |

**(ب) نمایش ترتیبی Sequential**

**شکل ۲۹ـ ۷**

## مثال ۲۰ـ ۷

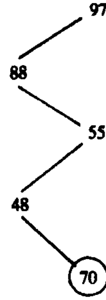
**Heap H** شکل ۲۹ـ ۷ را درنظر بگیرید. فرض کنید بخواهیم **ITEM = 70** را به **H** اضافه کنیم. نخست **70** را به عنوان عنصر بعدی در درخت کامل اضافه می‌کنیم، یعنی قرار می‌دهیم **TREE[21] = 70** آنگاه **70** بچهٔ راست **TREE[10] = 48** است. مسیر از **70** تا ریشهٔ **H** در شکل ۳۰ـ ۷ (الف) به تصویر درآمده است. اکنون مکان مربوط به **70** را در **Heap** به شرح زیر پیدا می‌کنیم:

(الف) **70** را با پدرش، **48** مقایسه کنید. از آنجا که **70** بزرگتر از **48** است، جای **70** و **48** را عوض کنید، مسیر اکنون به شکل ۳۰ـ ۷ (ب) درخواهد آمد.

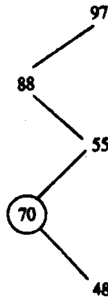
(ب) **70** را با پدر جدیدش، **55** مقایسه کنید از آنجا که **70** بزرگتر از **55** است، جای **70** و **55** را عوض کنید، مسیر اکنون به شکل ۳۰ـ ۷ (ج) درخواهد آمد.

(ج) **70** را با پدر جدیدش، **88** مقایسه کنید. از آنجا که **70** بزرگتر از **88** نیست، **ITEM = 70** به مکان مربوطه‌اش در **H** رسیده است.

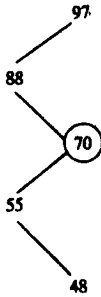
شکل ۳۰ـ ۷ (د) درخت نهایی را نشان می‌دهد. خط‌چین بیانگر آن است که یک جابجایی صورت گرفته است.



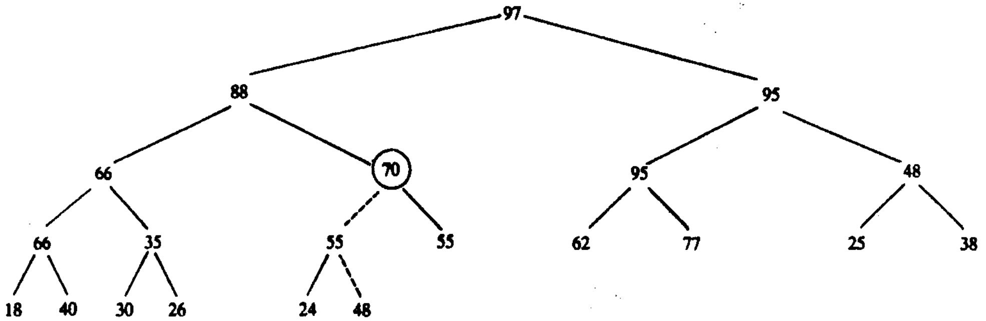
(الف)



(ب)



(ج)



(د)

**شکل ۳۰ـ ۷ ITEM = 70 اضافه می‌شود.**

**توجه کنید : می‌توان تحقیق کرد که زیربرنامهٔ Procedure بالا در نهایت همیشه منتهی بـه یک درخت Heap می‌شود. به عبارت دیگر هیچ چیز دیگری اتفاق نمی‌افتد. این مطلب به سادگی ثابت می‌شود و ما اثبات آن را به عنوان تمرین به دانشجو واگذار می‌کنیم.**

**مثال ۲۱ـ ۷**

**فرض کنید بخواهیم یک Heap H از لیست عددی زیر بسازیم :**

**44, 30, 50, 22, 60, 55, 77, 55**

**این کار را می‌توان با اضافه‌کردن هشت عدد یکی پس از دیگری در Heap H خالی با استفاده از زیربرنامهٔ Procedure بالا اضافه کرد. شکل از ۳۱ـ ۷ (الف) تا (ح) تصویرهای مربوطهٔ Heap را پس از اضافه‌شدن هر یک از هشت عنصر نشان می‌دهد. مجدداً خط چین بیانگر آن است که درطی اضافه‌کردن عنصر**