دارد، مقادیر پارامترها و متغیرهای محلی برای اجرای جاری P از پشته‌ها برگردانده می‌شود. کار با آدرسهای بازگشتی بسیار پیچیده است و به صورت زیر انجام می‌شود :

فرض کنید زیربرنامهٔ P شامل یک فراخوانی بازگشتی P در مرحلهٔ K باشد. آنگاه دو آدرس بازگشتی متناظر با اجرای این مرحلهٔ K وجود دارد.

(۱) آدرس بازگشتی جاری زیربرنامهٔP وجود دارد و هنگامی مورد استفاده قرار می‌گیرد که اجرای سطح جاری از اجرای P به پایان رسد.

(۲) آدرس بازگشتی جدید K+1 وجود دارد که آدرس مرحلهٔ بعد از احضار P است و برای بازگشت به سطح جاری از اجرای زیربرنامهٔ P مورد استفاده قرار می‌گیرد.

برخی از کتابها، اولین آدرس از این دو آدرس یعنی آدرس بازگشتی جاری، را به داخل آدرس بازگشتی پشته STADD ، Push می‌کنند در حالی که برخی دیگر، آدرس دوم یعنی آدرس بازگشتی جدید K+1 را به داخل پشتهٔ STADD ، Push می‌کنند. ما روش اول را انتخاب می‌کنیم چون تبدیل P به یک زیربرنامهٔ غیربازگشتی ساده‌تر خواهد بود. به‌خصوص به این تعبیر که یک پشتهٔ خالی STADD ، یک بازگشت به برنامهٔ اصلی را بیان می‌کند که در ابتدای کار، زیربرنامهٔ بازگشتی P را احضار می‌کند. تبدیل دیگری که آدرس بازگشتی جاری را به داخل پشته Push می‌کند در مسأله ۲۰ـ ۶ مورد بررسی قرار می‌گیرد.

الگوریتمی که زیربرنامهٔ بازگشتی P را به صورت یک زیربرنامهٔ غیربازگشتی تبدیل می‌کند به شرح زیر است: این الگوریتم از سه قسمت تشکیل شده است :

(۱) آماده‌سازی (۲) تبدیل فراخوانی بازگشتی P در زیربرنامهٔ P و (۳) تبدیل هر بازگشت Return در زیربرنامهٔ P .

(۱) "آماده‌سازی."

(الف) برای هر پارامتر STPAR یک پشتهٔ PAR ، برای هر متغیر محلی STVAR یک پشتهٔ VAR و برای نگهداری آدرسهای بازگشتی یک متغیر محلی ADD و یک پشتهٔ STADD تعریف کنید.

(ب) قرار دهید : TOP := NULL

(۲) تبدیل "مرحلهٔ K ، فراخوانی P"

(الف) مقادیر جاری پارامترها و متغیرهای محلی را به داخل پشته‌های مربوط Push کنید و آدرس بازگشت جدید [مرحله] K+1 را به داخل STADD ، Push کنید.

(ب) با استفاده از مقادیر آرگومان جاری پارامترها را مجدداً مقداردهی کنید.

(ج) به مرحلهٔ 1 بروید. [شروع زیربرنامهٔ P] .

(۳) تبدیل "مرحلهٔ J ، بازگشت Return"

(الف) اگر **STADD** خالی است، آنگاه بازگشت کنید **Return** . [کنترل کار به برنامهٔ اصلی بازگشت پیدا می‌کند.]

(ب) مقادیر بالای پشته‌ها را برگردانید، یعنی پارامترها و متغیرهای محلی را برابر مقادیر بالای پشته‌ها قرار دهید و **ADD** را برابر مقدار بالای پشتهٔ **STADD** قرار دهید.

(ج) به مرحله **ADD** بروید.

ملاحظه می‌کنید که تبدیل "مرحلهٔ K ، فراخوانی P" بستگی به مقدار K ندارد اما تبدیل "مرحلهٔ J ، بازگشت Return" به مقدار J بستگی دارد. بنابراین تنها لازم است که دستور بازگشت **Return** تبدیل شود برای مثال، با استفاده از

Step L. Return

مانند بالا و آنگاه

Go To StepL.

را جایگزین هر دستور بازگشت **Return** دیگر کنید.

این کار تبدیل زیربرنامهٔ **Procedure** را ساده می‌کند.

برجهای هانوی، صورت تجدیدنظر شده

بار دیگر مسأله برجهای هانوی را درنظر بگیرید. زیربرنامهٔ ۹ـ ۶ یک راه حل بازگشتی برای این مسأله با **n** دیسک است. ما این زیربرنامه را به یک حل غیربازگشتی تبدیل می‌کنیم. برای حفظ و نگهداری مراحل مشابه، دستور شروع را همانند مرحلهٔ **0** با **TOP := NULL** برچسب‌گذاری می‌کنیم. علاوه براین، تنها دستور بازگشت **Return** را در مرحلهٔ **5** همانند (۳) در صفحهٔ قبل تبدیل می‌کنیم.

Procedure 6.10: TOWER(N, BEG, AUX, END)
This is a nonrecursive solution to the Towers of Hanoi problem for N disks which is obtained by translating the recursive solution. Stacks STN, STBEG, STAUX, STEND and STADD will correspond, respectively, to the variables N, BEG, AUX, END and ADD.

0. Set TOP := NULL.
1. If N = 1, then:
   (*a*) Write: BEG → END.
   (*b*) Go to Step 5.
   [End of If structure.]
2. [Translation of "Call TOWER(N − 1, BEG, END, AUX)."]
   (*a*) [Push current values and new return address onto stacks.]
      (i) Set TOP := TOP + 1.
      (ii) Set STN[TOP] := N, STBEG[TOP] := BEG, STAUX[TOP] := AUX, STEND[TOP] := END, STADD[TOP] := 3.
   (*b*) [Reset parameters.]
   Set N := N − 1, BEG := BEG, AUX := END, END := AUX.
   (*c*) Go to Step 1.
3. Write: BEG → END.
4. [Translation of "Call TOWER(N − 1, AUX, BEG, END)."]
   (*a*) [Push current values and new return address onto stacks.]
      (i) Set TOP := TOP + 1.
      (ii) Set STN[TOP] := N, STBEG[TOP] := BEG, STAUX[TOP] := AUX, STEND[TOP] := END, STADD[TOP] := 5.
   (*b*) [Reset parameters.]
   Set N := N − 1, BEG := AUX, AUX := BEG, END := END.
   (*c*) Go to Step 1.
5. [Translation of "Return."]
   (*a*) If TOP := NULL, then: Return.
   (*b*) [Restore top values on stacks.]
      (i) Set N := STN[TOP], BEG := STBEG[TOP], AUX := STAUX[TOP], STEND[TOP], ADD := STADD[TOP].
      (ii) Set TOP := TOP − 1.
   (*c*) Go to Step ADD.

فرض کنید برنامهٔ اصلی شامل دستور زیر است :

Call TOWER(3, A, B, C)

اجرای حل مسأله در زیربرنامهٔ **6.10** را شبیه‌سازی می‌کنیم، تأکید ما در سطح‌های مختلف اجرای زیربرنامه است. هر سطح اجرا با مرحلهٔ مقدار اولیه دادن شروع می‌شود که در پارامترها، مقادیر آرگومانها از دستور فراخوانندهٔ اولیه یا از فراخوان بازگشتی در مرحلهٔ (۲) یا مرحلهٔ (۴) جایگزین می‌شود. از این‌رو هر آدرس بازگشت جدید یا مرحله ۳ یا مرحلهٔ ۵ است. شکل ۱۴ـ ۶ مرحله‌های مختلف پشته‌ها را نشان می‌دهد.

STN:	3	3, 2	3	3, 2	3		3	3, 2	3	3, 2	3	
STBEG:	A	A, A	A	A, A	A		A	A, B	A	A, B	A	
STAUX:	B	B, C	B	B, C	B		B	B, A	B	B, A	B	
STEND:	C	C, B	C	C, B	C		C	C, C	C	C, C	C	
STADD:	3	3, 3	3	3, 5	3		5	5, 3	5	5, 5	5	
	(a)	(b)	(c)	(d)	(e)	(f)	(g)	(h)	(i)	(j)	(k)	(l)

شکل ۱۴ـ ۶ پشته‌ها برای TOWER(3, A, B, C)

(الف) (سطح 1) فراخوانی اولیه CALL TOWER(3, A, B, C) مقادیر زیر را در پارامترها جایگزین می‌کند:

N := 3, BEG := A, AUX := B, END := C

مرحلهٔ ۱. چون $N \neq 1$، کنترل به مرحلهٔ ۲ داده می‌شود.

مرحلهٔ ۲. این یک فراخوان بازگشتی است. از این‌رو مقادیر جاری متغیرها و آدرس بازگشت جدید (در مرحلهٔ ۳) به داخل پشته‌ها مانند شکل ۱۴ـ ۶ (الف) Push می‌شوند.

(ب) (سطح ۲) مرحله ۲ فراخوان بازگشتی [TOWER(N − 1, BEG, END, AUX)] Callمقادیرزیر را در پارامترها جایگزین می‌کند:

N := N − 1 = 2, BEG := BEG = A, AUX := END = C, END := AUX = B

مرحلهٔ ۱. چون $N \neq 1$ کنترل به مرحلهٔ ۲ داده می‌شود.

مرحلهٔ ۲. این یک فراخوان بازگشتی است. از این‌رو مقادیر جاری متغیرها و آدرس بازگشت جدید (مرحلهٔ ۳) به داخل پشته‌ها مانند شکل ۱۴ـ ۶ (ب) Push می‌شوند.

(ج) (مرحلهٔ ۳) مرحلهٔ ۲ فراخوان بازگشتی [TOWER(N − 1, BEG, END, AUX)] مقادیر زیر را در پارامترها جایگزین می‌کند:

N := N − 1 = 1, BEG := BEG = A, AUX := END = B, END := AUX = C

مرحلهٔ ۱. اکنون N = 1 عمل BEG ⟶ END انتقال زیر را پیاده‌سازی می‌کند:

A ⟶ C

اکنون کنترل به مرحلهٔ ۵ منتقل می‌شود. [برای بازگشت Return]

مرحلهٔ ۵. پشته‌ها خالی نیستند، از این‌رو مقادیر بالای پشته‌ها حذف می‌شوند و به شکل ۱۴ـ ۶ (ج) درمی‌آید و جایگزینی‌های زیر صورت می‌گیرد:

N := 2, BEG := A, AUX := C, END := B, ADD := 3

کنترل به سطح ۲ قبلی در مرحلهٔ ADD منتقل می‌شود.

(د) (سطح ۲) [فعال شدن در مرحلهٔ ADD = 3]

مرحلهٔ ۳. عمل END ⟶ BEG انتقال زیر را پیاده‌سازی می‌کند :

$$A \longrightarrow B$$

مرحلهٔ ۴. این یک فراخوان بازگشتی است. از این‌رو مقادیر جاری متغیرها و آدرس بازگشت جدید (مرحلهٔ ۵) به داخل پشته‌ها مانند شکل ۱۴ـ ۶ (د) Push می‌شوند. (ه) (سطح ۳) مرحلهٔ ۴ فراخوان بازگشتی [TOWER(N − 1, AUX, BEG, END)] مقادیر زیر را در پارامترها جایگزین می‌کند :

N := N − 1 = 1, BEG := AUX = C, AUX := BEG = A, END := END = B

مرحلهٔ ۱ . اکنون N = 1 عمل BEG ⟶ END انتقال زیر را پیاده‌سازی می‌کند:

$$C \longrightarrow B$$

آنگاه کنترل به مرحلهٔ ۵ منتقل می‌شود. [برای بازگشت Return]

مرحلهٔ ۵ . پشته‌ها خالی نیستند، از این‌رو مقادیر بالای پشته‌ها حذف می‌شوند و به شکل ۱۴ـ ۶ (ه) درمی‌آید و جایگزینی‌های زیر صورت می‌گیرد :

N := 2, BEG := A, AUX := C, END := B, ADD := 5

کنترل به سطح ۲ قبلی در مرحلهٔ ADD منتقل می‌شود.

(و) (سطح ۲) [فعال شدن در مرحلهٔ ADD = 5]

مرحلهٔ ۵. پشته‌ها خالی نیستند، از این‌رو مقادیر بالای پشته‌ها حذف می‌شوند و به شکل ۱۴ـ ۶ (و) درمی‌آید و جایگزینی‌های زیر صورت می‌گیرد:

N := 3, BEG := A, AUX := B, END := C, ADD := 3

کنترل به سطح ۱ قبلی در مرحلهٔ ADD منتقل می‌شود.

(ز) (سطح ۱) [فعال شدن در مرحلهٔ . ADD = 3]

مرحلهٔ ۳ . عمل END ⟶ BEG انتقال زیر را پیاده‌سازی می‌کند :

$$A \longrightarrow C$$

مرحلهٔ ۴ . این یک فراخوان بازگشتی است. از این‌رو مقادیر جاری متغیرها و آدرس بازگشت جدید (مرحلهٔ ۵) به داخل پشته‌ها مانند شکل ۱۴ـ ۶ (و) Push می‌شوند.

(ح) (سطح ۲) مرحلهٔ ۴ فراخوان بازگشتی [TOWER(N − 1, AUX, BEG, END)] مقادیر زیر را در پارامترها جایگزین می‌کند:

N := N − 1 = 2, BEG := AUX = B, AUX := BEG = A, END := END C

مرحلهٔ ۱ . چون N ≠ 1 ، کنترل به مرحلهٔ ۲ منتقل می‌شود.

مرحلهٔ ۲ . این یک فراخوان بازگشتی است. از این‌رو مقادیر جاری متغیرها و آدرس بازگشت جدید (مرحلهٔ ۳) به داخل پشته‌ها مانند شکل ۱۴ـ ۶ (ح) **Push** می‌شوند.

(ط) (مرحلهٔ ۳) مرحلهٔ ۲ فراخوان بازگشت **[TOWER(N − 1, BEG, END, AUX)]** مقادیر زیر را در پارامترها جایگزین می‌کند:

N := N − 1 = 1, BEG := BEG = B, AUX := END = C, END := AUX = A

مرحلهٔ ۱. اکنون **N = 1** عمل **BEG ⟶ END** انتقال زیر را پیاده‌سازی می‌کند :

B ⟶ A

آنگاه کنترل به مرحلهٔ ۵ داده می‌شود. [برای بازگشت **Return**]

مرحلهٔ ۵ . پشته‌ها خالی نیستند از این‌رو مقادیر بالای پشته‌ها حذف می‌شوند و به شکل ۱۴ـ ۶ (ط) درمی‌آید و جایگزینی‌های زیر صورت می‌گیرد :

N := 2, BEG := B, AUX := A, END := C, ADD := 3

کنترل به سطح 2 قبلی در مرحلهٔ **ADD** منتقل می‌شود.

(ی) (سطح ۲) [فعال شدن در مرحلهٔ **ADD = 3**]

مرحلهٔ ۳ . عمل **BEG ⟶ END** انتقال زیر را پیاده‌سازی می‌کند :

B ⟶ C

مرحلهٔ ۴ . این یک فراخوان بازگشتی است. از این‌رو مقادیر جاری متغیرها و آدرس بازگشت جدید (مرحلهٔ ۵) به داخل پشته‌ها مانند شکل ۱۴ـ ۶ (ی) **Push** می‌شوند.

(ک) (سطح ۳) مرحله ۴ فراخوان بازگشتی **[OWER(N − 1, AUX, BEG, END)]** دیر زیر را در پارامترها جایگزین می‌کند :

N := N − 1 = 1, BEG := AUX = C, AUX := BEG = B, END := END = C

مرحلهٔ ۱. اکنون **N = 1** عمل **BEG ⟶ END** انتقال زیر را پیاده‌سازی می‌کند :

A ⟶ C

آنگاه کنترل به مرحلهٔ ۵ داده می‌شود. [برای بازگشت **Return**]

مرحلهٔ ۵ . پشته‌ها خالی نیستند از این‌رو مقادیر بالای پشته‌ها حذف می‌شوند و به شکل ۱۴ـ ۶ (ک) درمی‌آید و جایگزینی‌های زیر صورت می‌گیرد :

N := 2, BEG := B, AUX := A, END := C, ADD := 5

کنترل به سطح ۲ قبلی در مرحلهٔ **ADD** منتقل می‌شود.

(ل) (سطح ۲) [فعال شدن در مرحلهٔ **ADD = 5**]

مرحلهٔ ۵. پشته‌ها اکنون خالی نیستند. از این رو مقادیر بالای پشته‌ها حذف می‌شوند و به شکل ۱۶ـ ۴ (ل) در می‌آید و جایگزینی‌های زیر صورت می‌گیرد:

N := 3, BEG := A, AUX := B, END := C, ADD := 5

کنترل به سطح ۱ قبلی در مرحلهٔ ADD منتقل می‌شود.

(م) (سطح ۱) [فعال شدن در مرحلهٔ 5 = ADD]

مرحلهٔ ۵. پشته‌ها اکنون خالی هستند. بنابراین کنترل به برنامهٔ اصلی اولیه که شامل دستور زیر است داده می‌شود:

Call TOWER(3, A, B, C)

ملاحظه می‌کنید که خروجی شامل هفت انتقال زیر است:

$A \longrightarrow C,\quad A \longrightarrow B,\quad C \longrightarrow B,\quad A \longrightarrow C,\quad B \longrightarrow A,\quad B \longrightarrow C,\quad A \longrightarrow C,$

این نتیجه با جواب شکل ۱۱ـ ۶ سازگار است.

جمع‌بندی مطالب

مسأله برجهای هانوی، قدرت زیربرنامه‌های بازگشتی را در حل مسائل الگوریتمیِ متعدد روشن می‌سازد. این بخش چگونگی پیاده‌سازی زیربرنامه‌های بازگشتی را به وسیلهٔ پشته‌ها نشان می‌دهد و این درحالی است که از یک زبان برنامه‌نویسی نظیر **FORTRAN** یا **COBOL** استفاده می‌کنیم که نوشتن برنامه های بازگشتی در آنها مجاز نیست. درواقع، حتی وقتی که از یک زبان برنامه‌نویسی نظیر **PASCAL** استفاده می‌کنیم که از برنامه‌های بازگشتی پشتیبانی می‌کند، برنامه‌نویس ممکن است بخواهد از راه حل غیربازگشتی استفاده کند چون راه حل غیربازگشتی هزینهٔ کمتری نسبت به راه حل بازگشتی دارد.

۹ـ ۶ صفها

یک صف، لیست خطی از عناصر است که در آن عمل حذف تنها می‌تواند از یک انتهای آن موسوم به **سر صف یا ابتدای صف front** و عمل اضافه‌شدن تنها می‌تواند از انتهای دیگر آن موسوم **تهِ صف یا انتهای** آن **rear** صورت گیرد. اصطلاح ابتدای صف **front** و انتهای صف **rear** در توصیف یک لیست خطی تنها وقتی مورد استفاده قرار می‌گیرد که به عنوان یک صف پیاده‌سازی شود.

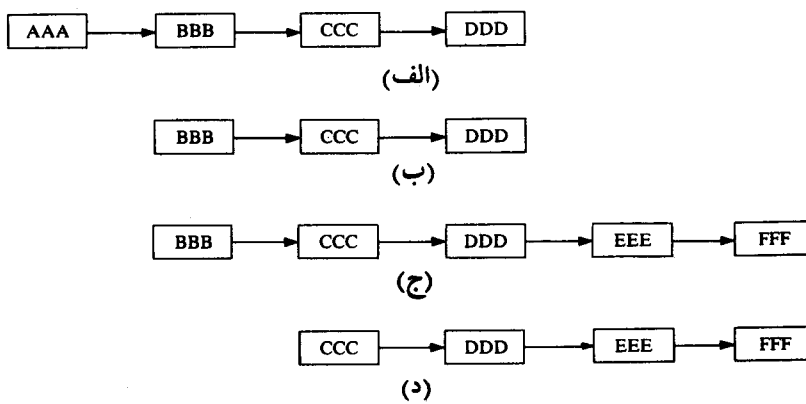
صفها، لیستهای **اولین ورودی اولین خروجی** است، **FIFO** نیز نامیده می‌شود چون اولین عنصر داخل یک صف، اولین عنصری است که از صف خارج می‌شود یا آن را ترک می‌کند. به بیان دیگر، ترتیبی که در آن عنصرها وارد یک صف می‌شوند به همان ترتیبی است که عنصرها، صف را ترک می‌کنند. صفها درمقابل پشته‌ها قرار دارند که لیستهای آخرین ورودی اولین خروجی است **LIFO** هستند.

صفها در زندگی روزمرهٔ ما به وفور دیده می‌شود. اتومبیلهایی که در یک چهارراه در انتظار عبور هستند تشکیل یک صف را می‌دهند که در آن، اولین اتومبیل وارد شده در صف انتظار، اولین اتومبیلی است که صف را ترک می‌کند. انسانهایی که در یک بانک در یک خط درانتظار انجام کارشان هستند، تشکیل یک صف را می‌دهند که در آن اولین نفر داخل خط، اولین کسی است که در انتظار انجام کارش به‌سر می‌برد و پس از انجام، اولین کسی است که صف را ترک می‌کند و الی آخر. یک مثال کامل از صف در علم کامپیوتر، در سیستم اشتراک زمانی اتفاق می‌افتد، که در آن برنامه‌هایی که دارای اولویت یکسان

هستند تشکیل یک صف را می‌دهند و در حال انتظار اجرا بسر می‌برند. ساختمان دیگری که از صف استفاده می‌کند یک صف **اولویت** نام دارد که در بخش ۱۱ـ ۶ مورد بحث و بررسی قرار خواهد گرفت.

مثال ۹ـ ۶

شکل ۱۵ـ ۶ (الف) نمودار یک صف با 4 عنصر را نشان می‌دهد که در آن **AAA** عنصر ابتدای صف و **DDD** عنصر انتهای صف است. ملاحظه می‌کنید که عنصر ابتدا و انتهای صف نیز به ترتیب عنصر اول و آخر لیست هستند. فرض کنید یک عنصر از صف حذف شده است. آنگاه عنصر حذف شده باید **AAA** باشد. این عمل صف را در وضعیت شکل ۱۵ـ ۶ (ب) قرار می‌دهد که در آن **BBB** اکنون عنصر اول صف است.



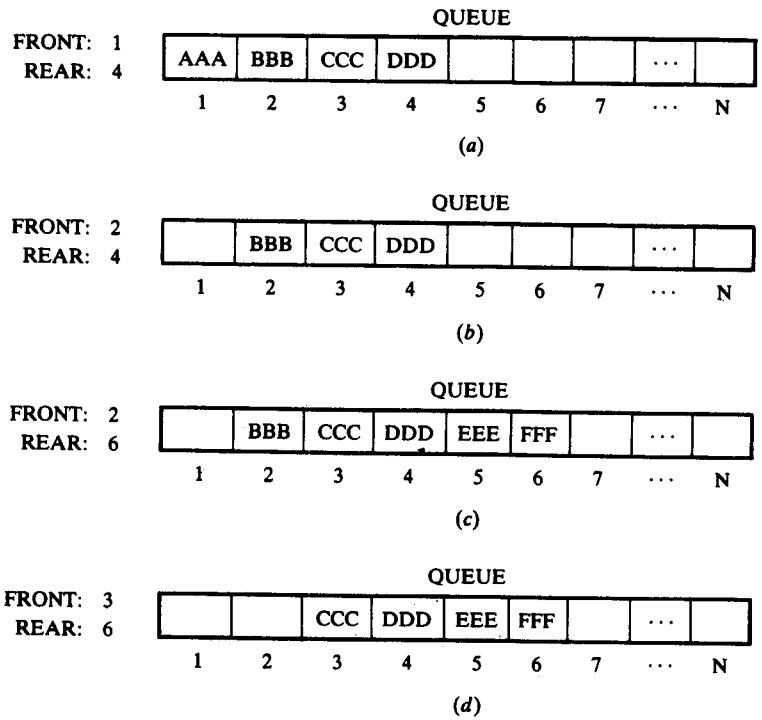
شکل ۱۵ـ ۶

بدنبال آن، فرض کنید **EEE** به صف اضافه شده است و آنگاه **FFF** به صف اضافه می‌شود. در آن صورت این عناصر باید به انتهای صف اضافه شوند. این وضعیت در شکل ۱۵ـ ۶ (ج) نشان داده شده است. توجه دارید که **FFF** اکنون عنصر انتهای صف است. حال فرض کنید عنصر دیگری از صف حذف شده است. آنگاه این عنصر باید **BBB** باشد که با این عمل صف در وضعیت شکل ۱۵ـ ۶ (د) قرار می‌گیرد و الی آخر. ملاحظه می‌کنید که در چنین ساختمان داده‌ای، **EEE** قبل از **FFF** حذف می‌شود چون این عنصر قبل از **FFF** در صف قرار گرفته است. بنابراین، **EEE** باید منتظر بماند تا **CCC** و **DDD** حذف شوند.

نمایش صفها

صفها را می‌توان در کامپیوتر به صورتهای مختلف نمایش داد اما معمولاً آنها را با لیستهای یکطرفه

یا آرایه‌های خطی نمایش می‌دهند. هر یک از صفهای داخل کتاب توسط یک آرایه خطی **QUEUE** و دو متغیر اشاره‌گرِ **FRONT** که شامل مکان عنصر ابتدای صف است و **REAR** که شامل مکان عنصر انتهای صف است پیاده‌سازی می‌شود مگر آن که خلاف آن به صورت صریح یا ضمنی بیان شود. شرط **FRONT = NULL** مبین آن است که صف خالی است.



شکل ۱۶ـ ۶ نمایش یک صف به وسیله آرایه

شکل ۱۶ـ ۶ روشی را نشان می‌دهد که در آن آرایهٔ شکل ۱۵ـ ۶ با استفاده از آرایهٔ **QUEUE** با **N** عنصر در حافظه ذخیره می‌شود. علاوه براین شکل ۱۶ـ ۶ روشی را نشان می‌دهد که عناصر از صف حذف می‌شوند و عناصر جدید به صف اضافه می‌شوند. ملاحظه می‌کنید که هرگاه یک عنصر از صف حذف شود، مقدار **FRONT** به اندازهٔ 1 افزایش می‌یابد و این حالت را می‌توان با دستور جایگزینی

FRONT := FRONT + 1

پیاده‌سازی کرد. به همین ترتیب، هرگاه یک عنصر به صف اضافه شود، مقدار **REAR** به اندازهٔ 1 افزایش می‌یابد و این حالت را می‌توان با دستور جایگزینی

REAR := REAR + 1

پیاده‌سازی کرد.

معنی آن، این است که پس از N عمل اضافه‌کردن، عنصر انتهای صف QUEUE[N] را اشغال می‌کند یا به بیان دیگر، نهایتاً صف آخرین قسمت آرایه را اشغال می‌کند. این وضعیت حتی وقتی خود صف عنصر زیاد نداشته باشد اتفاق می‌افتد.

فرض کنید بخواهیم عنصر ITEM را زمانی که صف آخرین قسمت آرایه را اشغال کرده است یعنی وقتی REAR = N است به صف اضافه کنیم. یک راه برای انجام این کار، آن است که تمام صف را به ابتدای آرایه منتقل کنیم و براساس آن FRONT و REAR را تغییر دهیم و آنگاه ITEM را مطابق آنچه که در بالا انجام دادیم به صف اضافه کنیم. این روش ممکن است پرهزینه و وقت‌گیر باشد. روشی که ما اتخاذ کرده‌ایم بر این فرض استوار است که QUEUE چرخشی یا حلقوی است یعنی QUEUE[1] پس از QUEUE[N] در آرایه قرار گرفته است. با این فرض با جایگزینی ITEM در QUEUE[1] عنصر ITEM را به صف اضافه می‌کنیم. به‌طور مشخص، بجای افزایش REAR به N + 1، قرار می‌دهیم REAR = 1 و آنگاه جایگزین می کنیم :

QUEUE[REAR] := ITEM

به همین ترتیب، اگر FRONT = N و یک عنصر از QUEUE حذف شود، بجای افزایش FRONT = 1 به FRONT قرار می‌دهیم N + 1 بعضی از دانشجویان این عملیات را به صورت حساب باقیمانده‌ای مورد توجه قرار می‌دهند که در بخش ۲ـ ۲ بررسی شده است.

فرض کنید صف تنها از یک عنصر تشکیل شده است یعنی فرض کنید که

FRONT = REAR ≠ NULL

و فرض کنید این عنصر حذف می‌شود. آنگاه جایگزینی‌های زیر را داریم :

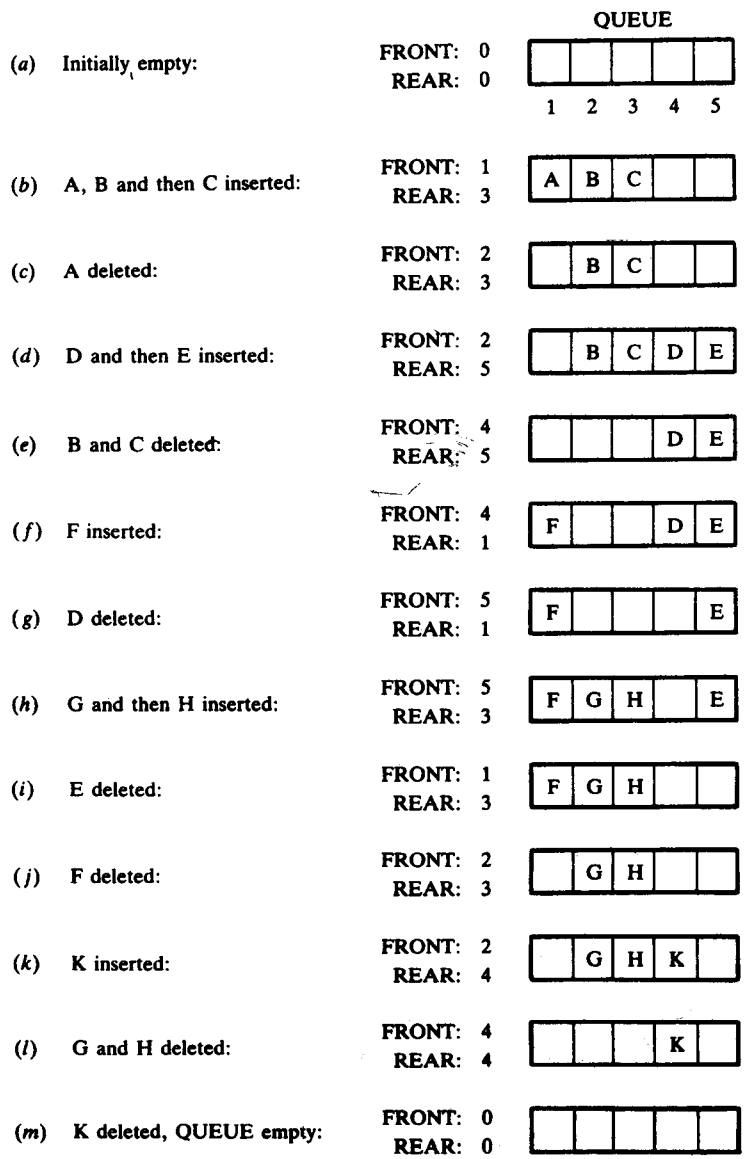
REAR := NULL و FRONT := NULL

که بیانگر آن است صف خالی است.

مثال ۱۰ـ ۶

شکل ۱۷ـ ۶ چگونگی پیاده‌سازی یک صف را توسط یک آرایهٔ چرخشی QUEUE با N = 5 خانهٔ حافظه نشان می‌دهد، ملاحظه می‌کنید که صف همیشه خانه‌های حافظهٔ متوالی را اشغال می‌کند بجز وقتی که، خانه‌هایی در ابتدا و انتهای آرایه را اشغال می‌کند. اگر صف را به صورت یک آرایهٔ چرخشی در نظر بگیریم معنی آن، این است که همچنان خانه‌های حافظه متوالی را اشغال می‌کنند. علاوه‌بر این، همانگونه که در شکل ۱۷ـ ۶ (م) مشاهده می‌شود، صف تنها وقتی خالی خواهد بود که

FRONT = REAR و یک عنصر حذف می‌شود. به همین دلیل NULL در FRONT و REAR در شکل ۱۷ـ ۶ (م) جایگزین می‌شود :



شکل ۱۷ـ ۶

اکنون وقت آن رسیده است تا زیربرنامهٔ **QINSERT** (زیربرنامهٔ ۱۱ـ ۶) را به صورت رسمی بیان کنیم که دادهٔ **ITEM** را به صف اضافه می‌کند. از اولین کارهایی که ما در این زیربرنامه انجام می‌دهیم آزمایش و کنترل وضعیت سرریزی است یعنی آزمایش کنیم که آیا صف پر است یا خیر؟

بدنبال آن زیربرنامهٔ **QDELETE** (زیربرنامهٔ **6.12**) را ارائه می‌دهیم که عنصر اول را از صف حذف می‌کند و آن را در متغیر **ITEM** جایگزین می‌کند. از اولین کارهایی که ما در زیربرنامه انجام می‌دهیم آزمایش و کنترل وضعیت زیرریزی است یعنی آزمایش کنیم که آیا صف خالی است یا خیر؟

Procedure 6.11: QINSERT(QUEUE, N, FRONT, REAR, ITEM)
This procedure inserts an element ITEM into a queue.

```
1. [Queue already filled?]
   If FRONT = 1 and REAR = N, or if FRONT = REAR + 1, then:
       Write: OVERFLOW, and Return.
2. [Find new value of REAR.]
   If FRONT := NULL, then: [Queue initially empty.]
       Set FRONT := 1 and REAR := 1.
   Else if REAR = N, then:
       Set REAR := 1.
   Else:
       Set REAR := REAR + 1.
   [End of If structure.]
3. Set QUEUE[REAR] := ITEM. [This inserts new element.]
4. Return.
```

Procedure 6.12: QDELETE(QUEUE, N, FRONT, REAR, ITEM)
This procedure deletes an element from a queue and assigns it to the variable ITEM.

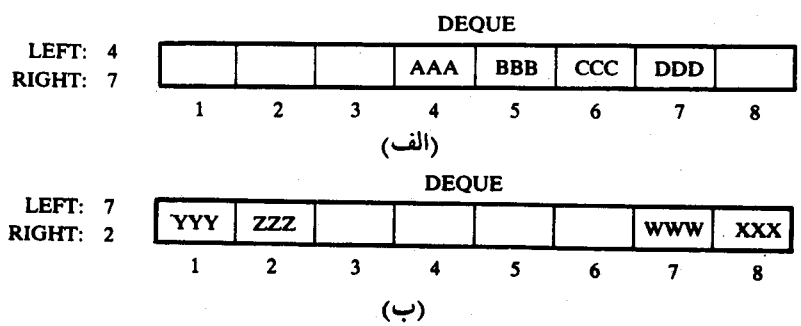
```
1. [Queue already empty?]
   If FRONT := NULL, then: Write: UNDERFLOW, and Return.
2. Set ITEM := QUEUE[FRONT].
3. [Find new value of FRONT.]
       If FRONT = REAR, then: [Queue has only one element to start.]
           Set FRONT := NULL and REAR := NULL.
       Else if FRONT = N, then:
           Set FRONT := 1.
       Else:
           Set FRONT := FRONT + 1.
   [End of If structure.]
4. Return.
```

۱۰ـ۶ صفهای دوسره یا DEQUE ها

یک **صف دوسره** یا یک **Deque** (به صورت دِکْ **Deck** یا دِکیو **Dequeue** تلفظ کنید) یک لیست خطی است که عناصر را می‌توان در آن از هر دو سر اضافه یا حذف کرد اما حذف یا اضافه کردن عنصر از وسط آن امکان‌پذیر نیست. اصطلاح **Deque** شکل اختصاری نام صف دوسره **Double – Ended Queue** است.

روشهای مختلفی برای نمایش یک **Deque** در کامپیوتر وجود دارد. فرض می‌کنیم که **Deque** ما توسط یک آرایهٔ چرخشی **DEQUE** با اشاره‌گرهای **LEFT** و **RIGHT** پیاده‌سازی می‌شود که به دوسر **Deque** اشاره می‌کند. مگر آن که خلاف آن به صورت صریح یا ضمنی بیان شود. فرض می‌کنیم که عناصر از سر سمت چپ تا سر سمت راست آرایه امتداد دارند. اصطلاح "چرخشی" یا "حلقوی" از این واقعیت گرفته شده است که فرض می‌کنیم عنصر **DEQUE[1]** پس از **DEQUE[N]** در آرایه می‌آید. شکل ۱۸ـ۶ دو **Deque** را نشان می‌دهد که هر یک با **4** عنصر در یک آرایهٔ **N = 8** خانهٔ حافظه‌ای پیاده‌سازی شده‌اند. شرط **LEFT = NULL** برای این منظور بکار می‌رود تا نشان دهد **Deque** خالی است.

دو نوع **Deque** وجود دارد که عبارتند از **Deque** با ورودی محدودشده و یک **Deque** با خروجی محدود شده که واسطهٔ بین یک **Deque** و یک صف هستند. یک **Deque** با **ورودی محدود شده** یک **Deque** است که به ما اجازه می‌دهد عمل اضافه‌کردن را تنها از یک سر لیست انجام دهیم اما در این نوع **Deque** ها مجاز هستیم از هر دو سر لیست عنصرها را حذف کنیم و یک **Deque** با **خروجی محدود شده** یک **Deque** است که به ما اجازه می‌دهد عمل حذف را تنها از یک سر لیست انجام دهیم اما در این نوع **Deque** ها مجاز هستیم از هر دو سر لیست عنصرها را اضافه کنیم.



شکل ۱۸ـ۶

این زیربرنامه‌های **Procedure** که عمل اضافه‌کردن و حذف عناصر را در **Deque** ها انجام می‌دهند و انواع مختلف اینگونه زیربرنامه‌ها به عنوان مسایل تکمیلی ارائه می‌شوند. همانند صفها، یک وضع

پیچیده ممکن است اتفاق بیفتد (الف) وقتی که سرریزی روی می‌دهد یعنی وقتی یک عنصر به Deque اضافه می‌شود که ازقبل پر است یا (ب) وقتی که زیرریزی روی می‌دهد یعنی وقتی یک عنصر از Deque حذف می‌شود که خالی است. زیربرنامه‌های Procedure باید این حالتهای ممکن را درنظر بگیرید.

۱۱ـ۶ صفهای اولویت

یک *صف اولویت*، یک مجموعه از عناصر است به طوری که به هر عنصر آن اولویت یکسان داده می‌شود و ترتیبی که در آن عناصر حذف و پردازش می‌شوند از دو قاعدهٔ زیر پیروی می کند :

(۱) عنصری که دارای اولویت بیشتر است قبل از تمام عناصری که اولویت کمتر دارد پردازش می‌شود.

(۲) دو عنصری که دارای اولویت یکسان هستند باتوجه به ترتیبی که به صف اضافه شده‌اند پردازش می‌شوند. یک نمونه از صف اولویت، یک سیستم اشتراک زمانی است. در این سیستم برنامه‌هایی که دارای اولویت بالاتر هستند اول پردازش می‌شوند و برنامه‌هایی که دارای اولویت یکسان هستند تشکیل یک صف استاندارد می‌دهند.

روشهای مختلفی برای پیاده‌سازی صف اولویت در حافظه وجود دارد. ما به شرح دو روش در اینجا می‌پردازیم. در یکی از این روشها از یک لیست یکطرفه و در روش دیگر از چند صف استفاده می‌شود. واضح است که میزان ساده یا مشکل‌بودن، اضافه‌کردن یک یا چند عنصر به صف اولویت یا حذف از آن به نوع نمایشی که برای صف انتخاب می‌کنیم بستگی دارد.

نمایش یک صف اولویت به وسیلهٔ لیست یکطرفه

یک راه پیاده‌سازی یک صف اولویت در حافظه به وسیلهٔ لیست یکطرفه است که به شرح زیر است :

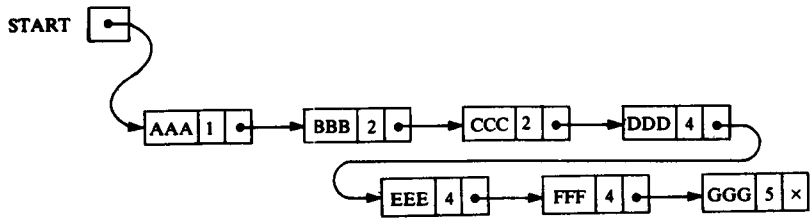
(الف) هر گره در لیست شامل سه عنصر اطلاعاتی است که عبارتند از: یک **فیلد اطلاعاتی INFO** ، یک **عدد اولویت PRN** و یک **عدد پیوند LINK** .

(ب) گره X قبل ازگرهٔ Y در لیست است (۱) اگر اولویت X بیشتر از اولویت Y باشد یا (۲) اگر هر دو گره اولویت یکسان دارند آنگاه X قبل از Y به لیست اضافه شده باشد، به بیان دیگر ترتیب در لیست یکطرفه متناظر با ترتیب صف اولویت است.

اعداد اولویت بر پایهٔ استنباط معمولی مورد استفاده قرار می‌گیرند، یعنی عناصری که عدد اولویت کوچکتری دارند، دارای اولویت بالاتری هستند.

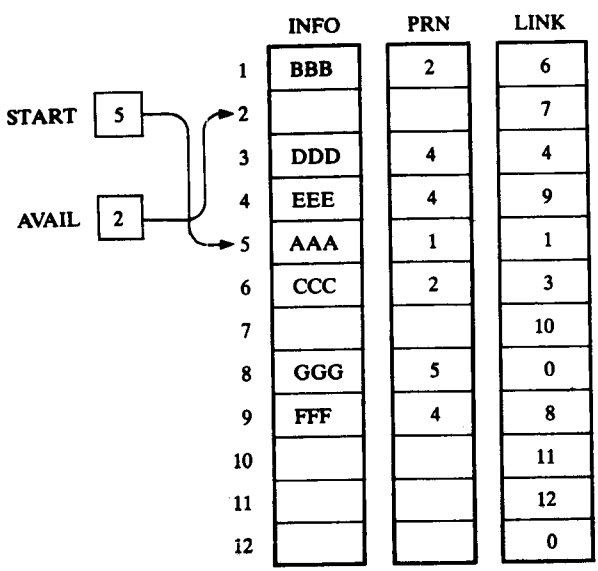
مثال ۱۱ـ۶

شکل ۱۹ـ۶ نمودار یک صف اولویت را با 7 عنصر نشان می‌دهد.



شکل ۱۹ـ ۶

نمودار در مورد این که **BBB** قبل یا بعد از **DDD** به لیست اضافه شده، چیزی به ما نمی‌گوید. از طرف دیگر، از نمودار استنباط می‌شود که **BBB** قبل از **CCC** به لیست اضافه شده است، چون **BBB** و **CCC** دارای عدد اولویت یکسان هستند و **BBB** قبل از **CCC** در لیست ظاهر شده است. شکل ۲۰ـ ۶ روشی را نشان می‌دهد که صف اولویت با استفاده از آرایه‌های خطی **INFO** و **PRN** و **LINK** (بخش ۲ـ ۵ را ببینید.) در حافظه ظاهر می‌شود.



شکل ۲۰ـ ۶

خاصیت اصلی نمایش یک صف اولویت با استفاده از لیست یکطرفه، آن است که عنصر داخل صف که همواره در ابتدای لیست یکطرفه ظاهر می‌شوند باید اول پردازش شوند. بنابراین عمل حذف یا اضافه‌کردن عنصر در صف اولویت بسیار ساده است. شرح این الگوریتم به صورت زیر است :

Algorithm 6.13: This algorithm deletes and processes the first element in a priority queue which appears in memory as a one-way list.

1. Set ITEM := INFO[START]. [This saves the data in the first node.]
2. Delete first node from the list.
3. Process ITEM.
4. Exit.

این الگوریتم اولین عنصر را از یک صف اولویت، که در حافظه به صورت لیست یکطرفه ظاهر شده است.حذف می‌کند و آنرا پردازش می‌کند.

جزئیات این الگوریتم، به همراه احتمال وقوع زیرریزی به عنوان تمرین به دانشجو واگذار می‌شود.

اضافه‌کردن یک عنصر به صف اولویت بسیار پیچیده‌تر از حذف یک عنصر از صف است چون برای اضافه‌کردن یک عنصر به صف احتیاج است جای درست آن را در صف پیدا کنیم. شرح این الگوریتم به صورت زیر است :

Algorithm 6.14: This algorithm adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

این الگوریتم عنصر ITEM با عدد اولویت N را در صف اولویت، که در حافظه به صورت لیست یکطرفه پیاده‌سازی شده است، اضافه می‌کند.

(الف) تا پیداشدن گرهٔ X که عدد اولویت آن بزرگتر از N است. لیست یکطرفه را پیمایش کنید ITEM را در جلوی گره X اضافه کنید.

(ب) اگر چنین گره‌ای پیدا نشد، ITEM را به عنوان آخرین عنصر لیست اضافه کنید.

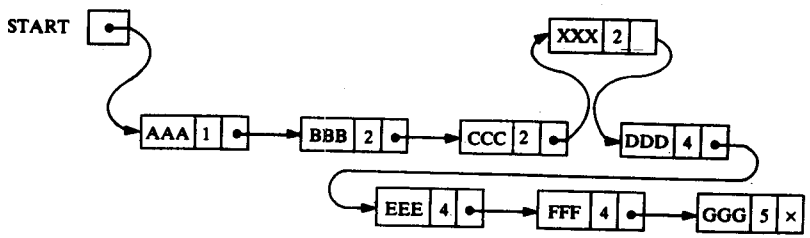
الگوریتم اضافه‌کردن بالا را می‌توان به صورت یک شیئی وزن داده شده که در لابلای عنصرها فرو می‌رود به تصویر درآورد تا به عنصری با وزن بیشتر برخورد کند.

جزئیات الگوریتم بالا به عنوان تمرین به دانشجو واگذار می‌شود. مشکل اصلی الگوریتم، از این واقعیت ناشی شده است که ITEM قبل از گرهٔ X اضافه می‌شود. به بیان دیگر هنگام پیمایش لیست، باید آدرس گره‌ای را که قبل از گرهٔ پردازش شونده قرار دارد نگهداریم.

مثال ۱۲ـ ۶

صف اولویت شکل ۱۹ـ ۶ را درنظر بگیرید. فرض کنید عنصر XXX با عدد اولویت 2 را می‌خواهیم به صف اضافه کنیم. با مقایسهٔ اعداد اولویت، لیست را پیمایش می‌کنیم. ملاحظه می‌کنید که

DDD عنصر اول لیست است که عدد اولویت آن بزرگتر از عدد اولویت **XXX** است. از این‌رو به صورتی که در شکل ۲۱ـ ۶ نشان داده شده است **XXX** در جلوی **DDD** در لیست اضافه می‌شود. ملاحظه می‌کنید که **XXX** پس از **BBB** و **CCC** ظاهر شده است که اولویت یکسان با **XXX** دارند. حال فرض کنید بخواهیم یک عنصر را از صف حذف کنیم. این عنصر **AAA** خواهد بود. که عنصر اول لیست است. با این فرض که هیچ عنصری به صف اضافه نمی‌شود، عنصر بعدی که حذف می‌شود **BBB** خواهد بود. آنگاه **CCC** و بدنبال آن **XXX** و الی آخر حذف خواهند شد.



شکل ۲۱ـ ۶

نمایش یک صف اولویت با استفاده از آرایه

یک روش دیگر برای پیاده‌سازی یک صف اولویت در حافظه، استفاده از یک صف جداگانه برای هر سطح از اولویت (یا برای هر عدد اولویت) است. هر یک از این صفها در آرایهٔ چرخشی یا حلقویِ مربوط به خود ظاهر می‌شود و باید جفت اشاره‌گر **FRONT** و **REAR** مختص خود را داشته باشند. درواقع چنانچه به هر صف مقدار مساوی اختصاص یابد، یک آرایهٔ دوبعدی **QUEUE** را می‌توان بجای آرایه‌های خطی مورد استفاده قرار داد. شکل ۲۲ـ ۶ این‌گونه نمایش صف اولویت را برای شکل ۲۱ـ ۶ نشان می‌دهد.

	FRONT	REAR
1	2	2
2	1	3
3	0	0
4	5	1
5	4	4

	1	2	3	4	5	6
1		AAA				
2	BBB	CCC	XXX			
3						
4	FFF				DDD	EEE
5				GGG		

شکل ۲۲ـ ۶

ملاحظه می‌کنید که FRONT[K] و REAR[K] به ترتیب شامل عناصر ابتدا و انتهای سطر K ام صف هستند. سطری که عناصر در آن دارای عدد اولویت K هستند.

در زیر شرحهایی از الگوریتم‌های حذف و اضافه‌کردن عناصر در یک صف اولویت ارائه شده است که مانند بالا به وسیلهٔ یک آرایهٔ دوبعدی QUEUE پیاده‌سازی می‌شود. جزئیات این الگوریتم‌ها به عنوان تمرین به دانشجو واگذار می‌شود.

Algorithm 6.15: This algorithm deletes and processes the first element in a priority queue maintained by a two-dimensional array QUEUE.

1. [Find the first nonempty queue.]
   Find the smallest K such that FRONT[K] ≠ NULL.
2. Delete and process the front element in row K of QUEUE.
3. Exit.

این الگوریتم عنصر اول را از یک صف اولویت که به صورت آرایهٔ دوبعدی QUEUE پیاده‌سازی شده است، حذف و آن را پردازش می‌کند.

Algorithm 6.16: This algorithm adds an ITEM with priority number M to a priority queue maintained by a two-dimensional array QUEUE.

1. Insert ITEM as the rear element in row M of QUEUE.
2. Exit.

این الگوریتم عنصر ITEM با عدد اولویت M را به یک صف اولویت که به صورت آرایهٔ دو بعدی QUEUE پیاده‌سازی شده است اضافه می‌کند.

خلاصهٔ مطالب فصل

هنگام انتخاب ساختمان داده‌های مختلف برای یک مسأله معین، بار دیگر توازن بین زمان و حافظه را ملاحظه کردید. نمایش یک صف اولویت به وسیلهٔ آرایه، از نظر زمان بسیار کاراتر از لیست یکطرفه است. علت آن هم این است که هنگام اضافه‌کردن یک عنصر به لیست یکطرفه، باید یک جستجوی خطی در لیست انجام داد. از طرف دیگر، نمایش صف اولویت به وسیلهٔ لیست یکطرفه می‌تواند ازنظر مصرف حافظه بسیار کاراتر از نمایش آرایه‌ای صف اولویت باشد، چون هنگام استفاده از نمایش آرایه‌ای، وقتی تعداد عناصر یک سطح اولویت از ظرفیت آن سطح بزرگتر می‌شود سرریزی اتفاق می‌افتد اما هنگام استفاده از لیست یکطرفه، سرریزی تنها وقتی اتفاق می‌افتد که تعداد کل عناصر بزرگتر از ظرفیت کل می‌شود. یک روش دیگر، استفاده از لیست پیوندی برای هر سطح اولویت است.

مسأله‌های حل‌شده

پشته‌ها

مسأله ۱ـ ۶ : پشته کاراکترهای زیر را درنظر بگیرید که در آن **STACK** به **N = 8** خانهٔ حافظه اختصاص داده می‌شود :

STACK: A, C, D, F, K, _, _, _

جهت سهولت در نمادگذاری از "ـ" برای نمایش خانهٔ حافظهٔ خالی استفاده می‌کنیم. وقتی عملیات زیر انجام می‌شود وضعیت پشته را بیان کنید :

(الف) **POP(STACK, ITEM)**
(ب) **POP(STACK, ITEM)**
(ج) **PUSH(STACK, L)**
(د) **PUSH(STACK, P)**
(ه) **POP(STACK, ITEM)**
(و) **PUSH(STACK, R)**
(ز) **PUSH(STACK, S)**
(ح) **POP(STACK, ITEM)**

حل : زیربرنامهٔ **POP** همیشه عنصر بالای پشته را حذف می‌کند و زیربرنامهٔ **PUSH** همیشه عنصر جدید به بالای پشته اضافه می‌کند. بنابراین :

(الف) **STACK: A, C, D, F, _, _, _, _**
(ب) **STACK: A, C, D, _, _, _, _, _**
(ج) **STACK: A, C, D, L, _, _, _, _**
(د) **STACK: A, C, D, L, P, _, _, _**
(ه) **STACK: A, C, D, L, _, _, _, _**
(و) **STACK: A, C, D, L, R, _, _, _**
(ز) **STACK: A, C, D, L, R, S, _, _**
(ح) **STACK: A, C, D, L, R, _, _, _**

مسأله ۲ـ ۶ : داده‌های مسأله ۱ـ ۶ را درنظر بگیرید. (الف) چه وقت سرریزی اتفاق می‌افتد؟ (ب) چه وقت **C** قبل از **D** حذف خواهد شد؟

حل : (الف) چون به **STACK** ، **N = 8** خانهٔ حافظه اختصاص داده شده است، سرریزی وقتی اتفاق می‌افتد که **STACK** حاوی **8** عنصر باشد و یک عمل **PUSH** برای اضافه‌کردن عنصر دیگر در **STACK** داشته باشیم.

(ب) چون **STACK** به صورت یک پشته پیاده‌سازی شده است، **C** هرگز قبل از **D** حذف نمی‌شود.

مسأله ۳ـ ۶ : پشتهٔ زیر را درنظر بگیرید که در آن به **STACK** ، **N = 6** خانهٔ حافظهٔ اختصاص داده شده است.

STACK: AAA, DDD, EEE, FFF, GGG, ____

وقتی عملیات زیر انجام می‌شود وضعیت پشته را شرح دهید :

(الف) PUSH(STACK, KKK), (ب) POP(STACK, ITEM),

(ج) PUSH(STACK, LLL), (د) PUSH(STACK, SSS),

(ه) POP(STACK, ITEM), (و) PUSH(STACK, TTT),

حل : (الف) KKK به بالای پشتهٔ STACK اضافه می‌شود، درنتیجه :

STACK: AAA, DDD, EEE, FFF, GGG, KKK

(ب) عنصر بالا از STACK حذف می‌شود، درنتیجه :

STACK: AAA, DDD, EEE, FFF, GGG, ____

(ج) LLL به بالای STACK اضافه می‌شود، درنتیجه :

STACK: AAA, DDD, EEE, FFF, GGG, LLL

(د) سرریزی اتفاق می‌افتد چون STACK پر است و عنصر دیگر SSS به STACK اضافه می‌شود.

هیچ یک از عملیات دیگر تا وقتی مسأله سرریزی حل نشود نمی‌توانند انجام شوند. عمل سرریزی را مثلاً می‌توان با افزودن حافظهٔ اضافی به STACK حل کرد.

مسأله ۴ـ ۶ : فرض کنید به STACK، n = 6 خانهٔ حافظه اختصاص داده شده است و در ابتدا STACK خالی است یا به بیان دیگر TOP = 0. خروجی قطعه برنامهٔ زیر را تعیین کنید.

```
1. Set AAA := 2 and BBB := 5.
2. Call PUSH(STACK, AAA).
   Call PUSH(STACK, 4).
   Call PUSH(STACK, BBB + 2).
   Call PUSH(STACK, 9).
   Call PUSH(STACK, AAA + BBB).
3. Repeat while TOP ≠ 0:
       Call POP(STACK, ITEM).
       Write: ITEM.
   [End of loop.]
4. Return.
```

مرحلهٔ ۱ . قرار دهید AAA = 2 و BBB = 5 .

مرحلهٔ ۲ . AAA = 2, 4, BBB + 2 = 7, 9 و AAA + BBB = 7 را در پشتهٔ STACK، Push کنید. نتیجه می‌شود :

STACK: 2, 4, 7, 9, 7, __

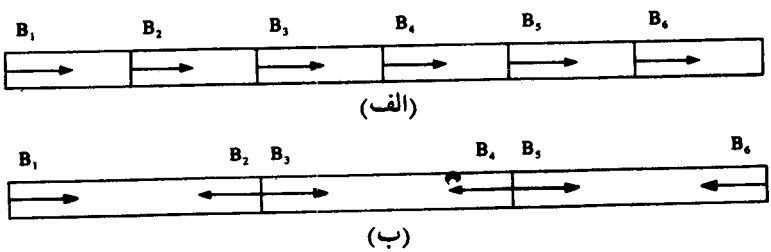
مرحلهٔ ۳ . عنصرهای STACK را تا وقتی STACK خالی نشده است POP کنید و چاپ نمائید. چون عنصر بالا همیشه POP می‌شود، خروجی از دنبالهٔ زیر تشکیل می‌شود :

7, 9, 7, 4, 2

ملاحظه می‌کنید که دنبالهٔ بالا به ترتیب عکس عناصری هستند که به **STACK** اضافه شده‌اند.

مسأله ۵ـ ۶ : فرض کنید به $K = 6$ پشته فضای معلوم S که از N خانهٔ حافظه همجوار تشکیل شده، اختصاص داده شده است. روشهای نگهداری پشته‌ها را در S شرح دهید.

حل : فرض کنید هیچ اطلاعاتی ازقبل در دست نیست تا بیان کند یک پشته خیلی سریع‌تر از پشتهٔ دیگر رشد می‌کند. آنگاه می‌توان N / K خانه برای هر پشته درنظر گرفت این عمل در شکل ۲۳ـ ۶ (الف) نشان داده شده است که در آن $B_1, B_2, ..., B_6$ به ترتیب عناصر پائین پشته‌ها را نمایش می‌دهند. به‌عبارت دیگر می‌توان پشته‌ها را به دو قسمت تقسیم کرد و $2N / K$ خانهٔ حافظه برای هر جفت پشته به صورتی که در شکل ۲۳ـ ۶ (ب) نشان داده شده است اختیار کرد. روش دوم می‌تواند تعداد دفعات وقوع سرریزی را کاهش دهد.



شکل ۲۳ـ ۶

نمادگذاری لهستانی

مسأله ۶ـ ۶ : با ملاحظه و بررسی و روش دستی هر عبارت میانوندی زیر را به عبارت پسوندی معادل آن تبدیل کنید :

(الف) $(A - B) * (D/E)$ (ب) $(A + B \uparrow D)/(E - F) + G$

(ج) $A * (B + D)/E - F * (G + H/K)$

حل : با استفاده از ترتیبی که با آن عملگرها اجرا می‌شوند، هر عملگر را از نماد میانوندی به پسوندی تبدیل می‌کنیم. از کروشهٔ باز و بسته [ ] برای نمایش بخشی از تبدیل انجام شده در هر مرحله استفاده می‌کنیم.

(الف) $(A - B) * (D/E) = [AB-] * [DE/] = AB-DE/*$

(ب) $(A + B \uparrow D)/(E - F) + G = (A + [BD\uparrow])/[EF-] + G = [ABD\uparrow+]/[EF-] + G$
$= [ABD\uparrow+EF-/] + G = ABD\uparrow+EF-/G+$

$$A*(B+D)/E-F*(G+H/K) = A*[BD+]/E-F*(G+[HK/]) \quad (ج)$$
$$= [ABD+*]/E-F*[GHK/+]$$
$$= [ABD+*E/]-[FGHK/+*]$$
$$= ABD+*E/FGHK/+*-$$

ملاحظه می‌کنید که تا وقتی عملوندها روی هم نیفتاده باشند ما در یک مرحله بیش از یک عملگر را به صورت پسوندی تبدیل کرده‌ایم.

مسأله ۷ـ ۶ : عبارت محاسباتی **P** زیر را که با نماد پسوندی نوشته شده است درنظر بگیرید :

P: 12, 7, 3, –, /, 2, 1, 5, +, * , +

(الف) با بررسی و روش دستی **P** را به عبارت میانوندی معادل آن تبدیل کنید.

(ب) عبارت میانوندی را ارزیابی کنید.

حل : (الف) عمل جستجو و خواندن را از چپ به راست انجام می‌دهیم، هر عملگر را از نماد پسوندی به میانوندی تبدیل می‌کنیم. از کروشهٔ باز و بسته [ ] برای نمایش بخشی از تبدیل انجام شده در هر مرحله استفاده می‌کنیم.

$$P = 12, [7-3], /, 2, 1, 5, +, *, +$$
$$= [12/(7-3)], 2, 1, 5, +, *, +$$
$$= [12/(7-3)], 2, [1+5], *, +$$
$$= [12/(7-3)], [2*(1+5)], +$$
$$= 12/(7-3)+2*(1+5)$$

(ب) با استفاده از عبارت میانوندی، به دست می‌آید :

$$P = 12/(7-3) + 2*(1+5) = 12/4 + 2*6 = 3 + 12 = 15$$

مسأله ۸ـ ۶ : عبارت پسوندی **P** مسأله ۷ـ ۶ را درنظر بگیرید. با استفاده از الگوریتم **6.3** ، **P** را ارزیابی کنید.

حل : نخست یک پرانتز بستهٔ نگهبان در انتهای **P** اضافه می‌کنیم به دست می‌آید :

P: 12, 7, 3, –, /, 2, 1, 5, +, * , +,)

عمل جستجو و خواندن **P** را از چپ به راست انجام می‌دهیم. اگر با یک مقدار ثابت روبرو شویم، آن را در پشته قرار می‌دهیم اما اگر با یک عملگر روبرو شویم دو مقدار ثابت بالای پشته را با آن عملگر ارزیابی می‌کنیم. شکل ۲۴ـ ۶ محتوی **STACK** را به‌محض جستجو و خوانده شدن هر عنصر **P** نشان می‌دهد.

Symbol	STACK
12	12
7	12, 7
3	12, 7, 3
−	12, 4
/	3
2	3, 2
1	3, 2, 1
5	3, 2, 1, 5
+	3, 2, 6
*	3, 12
+	15
)	15

شکل ۲۴ـ ۶

عدد آخر یعنی 15 در پشتهٔ STACK، وقتی پرانتز بستهٔ نگهبان خوانده می‌شود مقدار عبارت P است. این مقدار با نتیجهٔ مسأله ۷ـ ۶ (ب) سازگار است.

مسأله ۹ـ ۶ : عبارت میانوندی زیر Q را درنظر بگیرید :

Q: ((A + B) * D) ↑ (E − F)

با استفاده از الگوریتم 6.4، Q را به عبارت P پسوندی معادل آن تبدیل کنید.

حل : نخست یک پرانتز باز داخل پشتهٔ STACK، Push می‌کنیم و آنگاه درانتهای Q یک پرانتز بسته اضافه می‌کنیم به دست می‌آید :

Q: ((A + B) * D) ↑ (E − F))

توجه دارید که اکنون Q شامل 16 عنصر است. Q را از چپ به راست می‌خوانیم. یادآوری می‌کنیم که (۱) اگر به یک مقدار ثابت برخورد کنیم آن را به P اضافه می‌کنیم؛

(۲) اگر به یک پرانتز باز برخورد کنیم آن را به داخل پشته Push می‌کنیم (۳) اگر به یک عملگر برخورد کنیم آن را در سطح خودش پائین می‌بریم و (۴) اگر به یک پرانتز بسته برخورد کنیم آن را با اولین پرانتز باز فرو می‌بریم. شکل ۲۵ـ ۶ تصویرهای STACK و رشتهٔ P را به محض خوانده‌شدن هر عنصر Q نشان می‌دهد.

Symbol	STACK	Expression P
(	((	
(	(((	
A	(((	A
+	(((+	A
B	(((+	A B
)	((	A B +
*	((*	A B +
D	((*	A B + D
)	(	A B + D *
↑	(↑	A B + D *
(	(↑ (	A B + D *
E	(↑ (	A B + D * E
−	(↑ (−	A B + D * E
F	(↑ (−	A B + D * E F
)	(↑	A B + D * E F −
)		A B + D * E F − ↑

شکل ۲۵ـ ۶

هنگامی که **STACK** خالی است، پرانتز بستهٔ نهایی خوانده می‌شود و نتیجه چنین است :

P: A B + D * E F − ↑

که نماد پسوندی موردنظر معادل **Q** است.

مسأله ۱۰ـ ۶: با بررسی و روش دستی، هر عبارت میانوندی زیر را به عبارت پیشوندی معادل آن تبدیل کنید.

(الف) $(A - B) * (D / E)$

(ب) $(A + B \uparrow D)/(E - F) + G$

آیا هیچ رابطه‌ای بین عبارتهای پیشوندی و عبارتهای پسوندی معادل آن که در مسأله ۶ـ ۶ به دست آوردید وجود دارد؟

حل : با استفاده از ترتیبی که با آن عملگرها اجرا می‌شوند، هر عملگر را از نماد میانوندی به نماد پیشوندی تبدیل می‌کنیم.

(الف) $(A - B) * (D/E) = [-AB] * [/DE] = * - A B / D E$

(ب)

$$\begin{aligned}(A + B \uparrow D)/(E - F) + G &= (A + [\uparrow BD])/[-EF] + G \\ &= [+A\uparrow BD]/[-EF] + G \\ &= [/+A\uparrow BD - EF] + G \\ &= + / + A \uparrow B D - E F G\end{aligned}$$

عبارت پیشوندی عکس عبارت پسوندی نیست. با وجود این، ترتیب عملوندهای A, B, D و E در قسمت (الف) و A, B, D, E, F و G در قسمت (ب) در هر سه عبارت پیشوندی، پسوندی و میانوندی یکسان است.

QUICKSORT

مسأله ۱۱ـ ۶ : فرض کنید S لیست زیر، از 14 کاراکتر الفبایی تشکیل شده است :

(D) A T A S T R U C T U R E (S)

فرض کنید کاراکترهای S به صورت الفبایی مرتب باشند. با استفاده از الگوریتم QuickSort مکان نهایی کاراکتر اول D را پیدا کنید.

حل : کار را با کاراکتر آخر S شروع می‌کنیم. لیست را از راست به چپ جستجو و می‌خوانیم تا کاراکتری را پیدا کنیم که ازنظر الفبایی قبل از D قرار دارد. این کاراکتر C است. جای D و C را عوض می‌کنیم، لیست زیر بدست می‌آید :

(C) A T A S T R U (D) T U R E S

کار را با C شروع می‌کنیم، لیست را به طرف D یعنی از چپ به راست می‌خوانیم تا کاراکتری را پیدا کنیم که ازنظر الفبایی بعد از D قرار دارد. این کاراکتر T است. جای D و T را عوض می‌کنیم، لیست زیر بدست می‌آید :

C A (D) A S (T) R U T T U R E S

کار را با T شروع می‌کنیم، لیست را به طرف D می‌خوانیم تا کاراکتری را پیدا کنیم که قبل از D قرار دارد. این کاراکتر A است. جای D و A را عوض می‌کنیم، لیست زیر به دست می‌آید :

C A (A) (D) S T R U T T U R E S

کار را با A شروع می‌کنیم، لیست را به طرف D می‌خوانیم تا کاراکتری را پیدا کنیم که بعد از D قرار دارد. چنین حرفی وجود ندارد. معنی آن، این است که D در مکان نهایی‌اش است. علاوه براین، حرفهای قبل از D تشکیل لیست کوچکی از تمام حرفهایی را می‌دهند که از نظر الفبایی قبل از D هستند همچنین حرفهای بعد از D تشکیل لیست کوچکی از تمام حرفهایی را می‌دهند که ازنظر الفبایی بعد از D هستند، به صورت زیر :

C A A (D) S T R U T T U R E S

Sublist Sublist

اکنون مرتب‌کردن S به مرتب‌کردن هر یک از دو لیست کوچک تبدیل می‌شود.

مسأله ۱۲ـ ۶ : فرض کنید S از $n = 5$ حرف زیر تشکیل شده باشد :

(A) B C D (E)

C تعداد مقایسه‌هایی را تعیین کنید که برای مرتب‌کردن S با روش **QuickSort لازم** است. در صورت وجود، چه نتیجه کلی می‌توان از آن بدست آورد؟

حل :کار را با E شروع می‌کنیم برای این که بفهمیم حرف اول تاکنون در جای درستش قرار گرفته است یا نه. به $n - 1 = 4$ مقایسه احتیاج است. اکنون مرتب‌کردن S به مرتب‌کردن لیست کوچک زیر با $n - 1 = 4$ حرف منتهی می‌شود.

A (B) C D (E)

کار را با E شروع می‌کنیم، برای اینکه بفهمیم حرف اول B در لیست کوچک ازقبل در جای درستش قرار گرفته است یا نه، به $n - 2 = 3$ مقایسه احتیاج است. اکنون مرتب‌کردن S به مرتب‌کردن لیست کوچک زیر با $n - 2 = 3$ حرف منتهی می‌شود. به همین ترتیب برای این که بفهمیم حرف C در جای درستش قرار گرفته است به $n - 3 = 2$ مقایسه احتیاج است و برای اینکه بفهمیم حرف D در جای درستش است به $n - 4 = 1$ مقایسه احتیاج است. از آنجا که تنها یک حرف باقی می‌ماند لیست اکنون مرتب شده است. روی هم‌رفته تعداد :

$$C = 4 + 3 + 2 + 1 = 10 \quad \text{مقایسه}$$

داریم. به همین ترتیب با استفاده از **QuickSort** :

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

مقایسه احتیاج است تا یک لیست n عنصری را وقتی ازقبل مرتب است مرتب کند. نشان داده می‌شود که این حالت، بدترین حالت را برای **QuickSort** تشکیل می‌دهد.

مسأله ۱۳ـ ۶ : الگوریتم **QuickSort** را درنظر بگیرید. **(الف)** آیا می‌توان آرایه‌های **LOWER** و **UPPER** را به عوض پشته به صورت صف پیاده‌سازی کرد؟ چرا؟ **(ب)** برای الگوریتم **Quicksort** به چه مقدار حافظهٔ اضافی موردنیاز است یا به عبارت دیگر، پیچیدگی الگوریتم از نظر حافظه چقدر است؟

حل : (الف) از آنجا که اهمیت ندارد زیرمجموعه‌ها با چه ترتیبی مرتب می‌شوند، از این‌رو **LOWER** و **UPPER** را می‌توان به عوض پشته با صف یا حتی با صف دوسره یا **Deque** پیاده‌سازی کرد.

(ب) الگوریتم QuickSort یک الگوریتم "درجا" است یعنی اینکه عناصر به استثنای حالتی که جایشان عوض می‌شوند درجای خودشان باقی می‌مانند. اساساً حافظهٔ اضافی برای پشته‌ها LOWER و UPPER موردنیاز است. درحالت میانگین، مقدار حافظهٔ اضافی موردنیاز برای این الگوریتم متناسب با $\log n$ است که در آن n تعداد عناصر مرتب شده است.

زیربرنامهٔ بازگشتی

مسأله ۱۴ـ ۶ : فرض کنید a و b نمایش دو عدد صحیح مثبت باشند. فرض کنید تابع Q به شکل زیر به صورت بازگشتی تعریف شده است :

$$Q(a, b) = \begin{cases} 0 & \text{if } a < b \\ Q(a - b, b) + 1 & \text{if } b \le a \end{cases}$$

(الف) مقدار Q(2, 3) و Q(14, 3) را پیدا کنید.

(ب) این تابع چه عملی انجام می‌دهد؟ مقدار Q(5861, 7) را پیدا کنید.

حل : (الف) چون $2 < 3$ $Q(2, 3) = 0$

$$\begin{aligned} Q(14, 3) &= Q(11, 3) + 1 \\ &= [Q(8, 3) + 1] + 1 = Q(8, 3) + 2 \\ &= [Q(5, 3) + 1] + 2 = Q(5, 3) + 3 \\ &= [Q(2, 3) + 1] + 3 = Q(2, 3) + 4 \\ &= 0 + 4 = 4 \end{aligned}$$

(ب) هر بار که b از a کم می‌شود مقادیر Q یک واحد افزایش می‌یابد. از این‌رو Q(a, b) وقتی a بر b تقسیم می‌شود خارج‌قسمت را پیدا می‌کند. بنابراین

$$Q(5861, 7) = 837$$

مسأله ۱۵ـ ۶ : فرض کنید n یک عدد صحیح مثبت باشد. فرض کنید تابع بازگشتی L به صورت زیر تعریف شده است.

$$L(n) = \begin{cases} 0 & \text{if } n = 1 \\ L(\lfloor n/2 \rfloor) + 1 & \text{if } n > 1 \end{cases}$$

در اینجا $\lfloor K \rfloor$ "کف floor" عدد K را نشان می‌دهد و بزرگترین عدد صحیحی است که بزرگتر از K نباشد. بخش ۲ـ ۲ را ببینید.

(الف) مقدار L(25) را بدست آورید.

(ب) این تابع چه عملی انجام می‌دهد؟

حل : (الف)

$$\begin{aligned} L(25) &= L(12) + 1 \\ &= [L(6) + 1] + 1 = L(6) + 2 \\ &= [L(3) + 1] + 2 = L(3) + 3 \\ &= [L(1) + 1] + 3 = L(1) + 4 \\ &= 0 + 4 = 4 \end{aligned}$$

(ب) هر بار که n بر 2 تقسیم می‌شود مقدار L یک واحد افزایش می‌یابد. از این‌رو L بزرگترین عدد صحیحی است که

$$2^L \leq n$$

بنابراین تابع

$$L = \lfloor \log_2 n \rfloor$$

را به دست می‌آورد.

مسأله ۱۶ـ ۶ : فرض کنید اعداد فیبوناچی $F_{11} = 89$ و $F_{12} = 144$ داده شده‌اند.

(الف) برای محاسبه F_{16} آیا باید از روش بازگشتی استفاده کنیم یا روش تکرار؟

مقدار F_{16} را پیدا کنید.

(ب) یک زیربرنامهٔ **Procedure** با روش تکرار بنویسید تا نخستین N عدد فیبوناچی F[1], F[2], ..., F[N] را به دست آورد که در آن N>2 این زیربرنامه را با زیربرنامهٔ بازگشتی **6.8** مقایسه کنید.

حل : (الف) به‌عوض استفاده از روش بازگشتی (که در آن عمل ارزیابی از بالا به پائین انجام می‌شود) بهتر است اعداد فیبوناچی با روش تکرار بدست آید (یعنی عمل ارزیابی از پائین به بالا انجام شود.)

یادآوری می‌کنیم که هر عدد فیبوناچی مجموع دو عدد فیبوناچی قبل از خودش است. با شروع از F_{11} و F_{12} داریم :

$$F_{13} = 89 + 144 = 233, \quad F_{14} = 144 + 233 = 377, \quad F_{15} = 233 + 377 = 610$$

و درنتیجه :

$$F_{16} = 377 + 610 = 987$$

(ب)

Procedure P6.16: FIBONACCI(F, N)

این زیربرنامه نخستین N عدد فیبوناچی را پیدا می‌کند و آنها را در آرایهٔ F قرار می‌دهد.

```
1. Set F[1] := 1 and F[2] := 1.
2. Repeat for L = 3 to N:
       Set F[L] := F[L − 1] + F[L − 2].
   [End of loop.]
3. Return.
```

تأکید می‌کنیم که این زیربرنامه با روش تکرار بسیار کاراتر از زیربرنامهٔ بازگشتی **6.8** است.

مسأله ۱۷ـ ۶ : با استفاده از تعریف تابع آکرمان (تعریف ۳ـ ۶) ، **A(1, 3)** را بدست آورید.

حل : ما 15 مرحلهٔ زیر را داریم :

```
(1)  A(1, 3) = A(0, A(1, 2))
(2)      A(1, 2) = A(0, A(1, 1))
(3)          A(1, 1) = A(0, A(1, 0))
(4)              A(1, 0) = A(0, 1)
(5)                  A(0, 1) = 1 + 1 = 2
(6)              A(1, 0) = 2
(7)          A(1, 1) = A(0, 2)
(8)              A(0, 2) = 2 + 1 = 3
(9)          A(1, 1) = 3
(10)     A(1, 2) = A(0, 3)
(11)         A(0, 3) = 3 + 1 = 4
(12)     A(1, 2) = 4
(13) A(1, 3) = A(0, 4)
(14)     A(0, 4) = 4 + 1 = 5
(15) A(1, 3) = 5
```

حالت پله‌ای به طرف جلو بیانگر آن است که یک ارزیابی را به‌تعویق انداختیم و می‌خواهیم از تعریف استفاده کنیم و حالت پله‌ای به عقب بیانگر آن است که می‌خواهیم از انتها به ابتدا برگردیم و ملاحظه می‌کنید که از فرمول اول تعریف 6.3 در مرحله‌های 5 ، 8 ، 11 و 14 استفاده شده است و از فرمول دوم در مرحلهٔ 4 و از فرمول سوم در مرحله‌های 1 ، 2 و 3 استفاده شده است. در مراحل دیگر با جایگزینی‌ها به ابتدا و عقب برمی‌گردیم.

مسأله ۱۸ـ ۶ : فرض کنید یک زیربرنامه بازگشتی **P** تنها یک بار فراخوانی بازگشتی دارد :

Step K. Call P.

دلیلی ارائه دهید که چرا به پشتهٔ **STADD** (برای آدرسهای بازگشتی) احتیاج نیست.

حل : از آنجا که تنها یک فراخوانی بازگشتی وجود دارد کنترل کار همیشه به مرحلهٔ **K+1** برای یک بازگشت **Return** منتقل می‌شود، به‌استثنای بازگشت نهایی که به برنامهٔ اصلی است. بنابراین به عوض نگهداری پشتهٔ **STADD** (و متغیر محلی **ADD**) فقط به جای

(c) Go to Step K + 1

در تبدیل " **Step J. Return** " می‌نویسیم. (بخش ۸ـ ۶ را ببینید) :

(c) Go to Step ADD

مسأله ۱۹ـ ۶ : راه حل مسأله برجهای هانوی را به‌گونه‌ای بازنویسی کنید که در آن به جای دو فراخوانی بازگشتی تنها از یک فراخوانی استفاده شود.

حل: یک راه آن است که میله‌های **A** و **B** را به صورت متقارن درنظر بگیریم، یعنی مراحل زیر را بکار

بندیم :

تعداد $N-1$ دیسک را از **A** به **B** منتقل کنید و آنگاه **A→C** را بکار بندید.

تعداد $N-2$ دیسک را از **B** به **A** منتقل کنید و آنگاه **B→C** را بکار بندید.

تعداد $N-3$ دیسک را از **A** به **B** منتقل کنید و آنگاه **A→C** را بکار بندید.

تعداد $N-4$ دیسک را از **B** به **A** منتقل کنید و آنگاه **B→C** را بکار بندید.

و الی آخر. بنابراین می‌توانیم تنها یک فراخوانی بازگشتی را تکرار کنیم که پس از هر تکرار جای **BEG** و **AUX** را عوض می‌کند به صورت زیر :

```
Procedure P6.19: TOWER(N, BEG, AUX, END)
    1. If N = 0, then: Return.
    2. Repeat Steps 3 to 5 for K = N, N - 1, N - 2, . . . , 1.
    3.     Call TOWER(K - 1, BEG, END, AUX).
    4.     Write: BEG → END.
    5.     [Interchange BEG and AUX.]
           Set TEMP := BEG, BEG := AUX, AUX := TEMP.
       [End of Step 2 loop.]
    6. Return.
```

ملاحظه می‌کنید که ما در اینجا به جای $N = 1$ از $N = 0$ به عنوان مقدار پایه برای بازگشت استفاده می‌کنیم.

مسأله ۲۰ـ ۶ : پیاده‌سازی پشته‌ای الگوریتم بخش ۸ـ ۶ را برای تبدیل یک زیربرنامهٔ بازگشتی به یک برنامهٔ غیربازگشتی درنظر بگیرید. یادآوری می‌کنیم که در زمان فراخوانی بازگشتی به جای آدرس بازگشت جاری، آدرس بازگشت جدید را به داخل پشتهٔ **STADD** ، **Push** می‌کنیم.

فرض کنید خواسته باشیم آدرس بازگشت جاری را به داخل پشتهٔ **STADD** ، **Push** کنیم. (در بسیاری از متنها و کتابها اینگونه عمل می‌کنند) چه تغییری لازم است در الگوریتم تبدیل داده شود؟

حل : تغییر اصلی عبارت است از آن که در زمان بازگشت به سطح اجرای قبلی، مقدار جاری **ADD** مکان بازگشت را تعیین می‌کند نه مقدار **ADD** را پس از **POP** شدن مقادیر پشته‌ها. بنابراین مقدار **ADD** باید با دستور **SAVE := ADD** نگهداری شود، آنگاه مقادیر پشته‌ها **POP** می‌شوند و بدنبال آن کنترل به مرحلهٔ **SAVE** منتقل می‌شود. تغییر دیگر آن است که باید از ابتدا دستور جایگزینی **ADD = Main** را داشته باشیم و وقتی **ADD = Main** به برنامهٔ فراخوانندهٔ اصلی بازگشت **Return** کنیم نه وقتی که پشته‌ها خالی هستند. الگوریتم به صورت رسمی به شرح زیر است :

(۱) "آماده‌سازی"

(الف) یک پشته **STPAR** برای هر پارامتر **PAR** ، یک پشته **STVAR** برای هر متغیر محلی **VAR** و یک متغیر محلی **ADD** و یک پشتهٔ **STADD** برای نگهداری آدرسهای بازگشتی تعریف کنید.