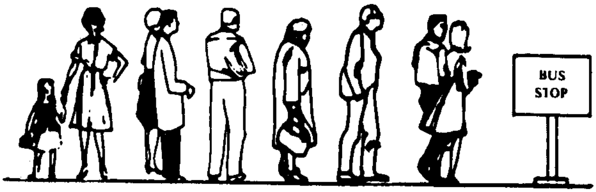
ملاحظه می‌کنید که یک قلم از عنصر را می‌توان تنها از بالای هر یک از این پشته‌ها حذف کرد یا به بالای آن اضافه کرد. به‌ویژه این که آخرین عنصر داده‌ای اضافه‌شده به یک پشته، اولین عنصر داده‌ای است که می‌توان از آن حذف کرد. به این ترتیب به پشته‌ها، لیستهای **آخرین ورودی اولین خروجی** است، LIFO نیز می‌گویند. نامهای دیگری برای پشته‌ها بکار برده می‌شود. که عبارتند از: "انبوه" و "لیستهای فشردنی" . هرچند ممکن است پشته ساختمان داده بسیار محدودی بنظر رسد اما در علم کامپیوتر کاربرد بسیار زیادی دارد.

یک **صف**، یک لیست خطی است که در آن هر عنصر داده‌ای را می‌توان تنها از یک انتهای آن اضافه کرد و عنصرهای داده‌ای را می‌توان تنها از انتهای دیگر آن حذف کرد. نام صف احتمالاً از کاربرد روزمرهٔ این اصطلاح گرفته شده است. صفِ مردمی را که در یک ایستگاه، به صورتی که در شکل ۲ـ ۶ می‌بینید، منتظر اتوبوس هستند را درنظر بگیرید.



شکل ۲ـ ۶ . صف انتظار اتوبوس

هر شخص جدید که وارد ایستگاه می‌شود در آخر صف می‌ایستد و وقتی اتوبوس می‌رسد مردمانی که در جلوی صف هستند اول، سوار اتوبوس می‌شوند. واضح است که اولین نفر داخل صف اولین کسی است که صف انتظار را ترک می‌کند. بنابراین به صفها لیستهای **اولین ورودی اولین خروجی** است، FIFO نیز می‌گویند. مثال دیگری از یک صف، یک تعداد یا یک بسته از برنامه‌های است که درانتظار پردازش بسر می‌برند. فرض می‌شود که هیچ برنامه‌ای اولویت بالاتری نسبت به دیگری ندارد.

مفهوم بازگشتی، از مفاهیم اساسی و بنیادیِ علم کامپیوتر است. این مبحث در این بخش معرفی می‌شود زیرا به وسیلهٔ ساختمان یک پشته می‌توان مفاهیم مربوط به مسایل بازگشتی را شبیه‌سازی کرد.

۲ـ ۶ پشته‌ها

یک **پشته**، یک لیست از عناصر است که در آن هر عنصر را می‌توان تنها از یک انتها موسوم به **بالای**

پشته حذف یا اضافه کرد، یعنی عناصر به ترتیب عکسی که وارد پشته می‌شوند از پشته حذف می‌شوند.

دو اصطلاح خاص، برای دو عمل اساسی، با پشته‌ها بکار می‌رود :

(الف) عمل **PUSH** که این اصطلاح برای اضافه‌کردن یک عنصر در پشته بکار می‌رود.

(ب) عمل **POP** که این اصطلاح برای حذف یک عنصر از پشته بکار می‌رود.

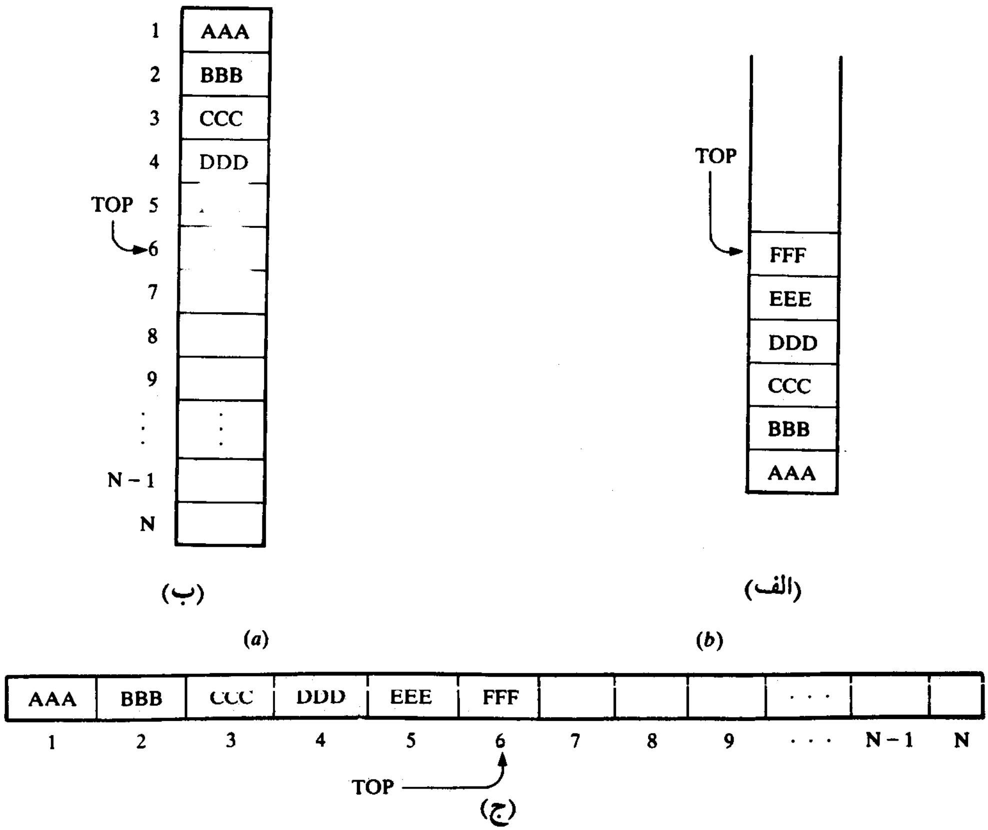
تأکید می‌کنیم که این اصطلاحات تنها هنگام کار با پشته‌ها بکار می‌روند و در هیچ ساختمان دادهٔ دیگری، مورد استفاده قرار نمی‌گیرد.

مثال ۱ـ ۶

فرض کنید 6 عنصر زیر به‌ترتیب در یک پشتهٔ خالی **PUSH** می‌شوند :

AAA, BBB, CCC, DDD, EEE, FFF

شکل ۳ـ ۶ سه راه به تصویر کشیده‌شدن چنین پشته‌ای را نشان می‌دهد.



شکل ۳ـ ۶ . نمودار پشته‌ها

جهت سهولت در نمادگذاری، اغلب پشته را به صورت زیر مشخص می‌کنیم:

STACK: AAA, BBB, CCC, DDD, EEE, FFF

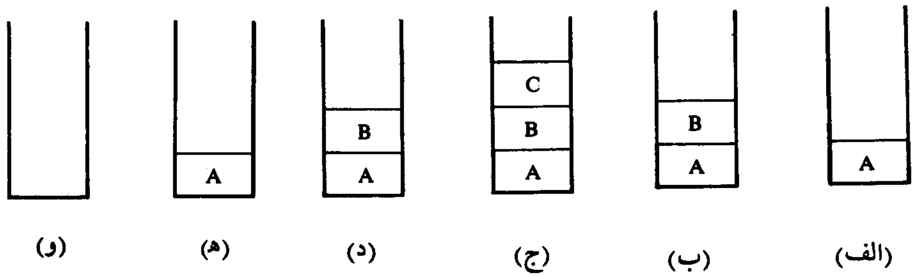
ضرورت این کار در آن است که سمت راست‌ترین عنصر، در بالای پشته قرار می‌گیرد. صرفنظر از روشی که یک پشته توصیف می‌شود، تأکید می‌کنیم که خاصیت اساسی آن، که همان عمل اضافه کردن و حذف عنصر است می‌تواند تنها در بالای پشته اتفاق بیفتد. معنی آن این است که قبل از حذف **FFF** نمی‌توان **EEE** را حذف کرد و قبل از حذف **EEE** و **FFF** نمی‌توان **DDD** را حذف کرد و الی آخر. درنتیجه عناصر را می‌توان از پشته، تنها به‌ترتیب عکسی که در پشته **PUSH** یا اضافه می‌شود، **POP** یا حذف کرد.

بار دیگر لیست گره‌های آزاد **AVAIL** را که در فصل ۵ بررسی کردیم درنظر بگیرید. یادآوری می‌کنیم که گره‌های آزاد تنها از ابتدای لیست **AVAIL** حذف می‌شوند و گره‌های جدید موجود تنها در ابتدای لیست **AVAIL** اضافه می‌شوند. به بیان دیگر لیست **AVAIL** به صورت یک پشته پیاده‌سازی می‌شود. این روش پیاده‌سازی لیست **AVAIL** به صورت یک پشته تنها بخاطر سادگی و سهولت آن نسبت به قسمت اصلی این ساختمان است. در قسمت زیر، وضعیت مهم و قابل توجهی را مورد بررسی قرار می‌دهیم که در آن پشته ابزار اساسی پردازش خود الگوریتم است.

تصمیم‌گیری‌های درجه دوم یا به‌تعویق افتاده

پشته‌ها اغلب برای بیان ترتیب مراحل پردازش‌هایی بکار می‌رود که در آن مراحل، یعنی پردازش باید تا برقراری و محقق‌شدن شرایط دیگر به تعویق بیفتد. به مثال زیر توجه کنید.

فرض کنید که هنگام پردازش پروژهٔ **A** نیازمند آن باشیم که روی پروژهٔ **B** کار کنیم که کامل‌شدن **B** مستلزم کامل‌شدن پروژهٔ **A** است. آنگاه پوشه‌ای که شامل داده‌های پروژهٔ **A** است را در پشته قرار می‌دهیم، این وضعیت در شکل ۴ـ ۶ (الف) به تصویر کشیده شده است، همچنین شروع به پردازش **B** می‌کنیم. با وجود این فرض کنید با همان دلیل هنگام پردازش **B** منتهی به پردازش پروژهٔ **C** می‌شویم. آنگاه همانند آنچه که در نمودار ۴ـ ۶ (ب) به تصویر درآوردیم **B** را در پشته بالای **A** قرار می‌دهیم و شروع به پردازش **C** می‌کنیم. علاوه براین فرض کنید هنگام پردازش **C** به همین ترتیب منتهی به پردازش **D** شویم. آنگاه **C** را در پشته بالای **B** قرار می‌دهیم، این وضعیت در شکل ۴ـ ۶ (ج) به تصویر کشیده شده است و همچنین شروع به پردازش **D** می‌کنیم.



شکل ۴ـ ۶

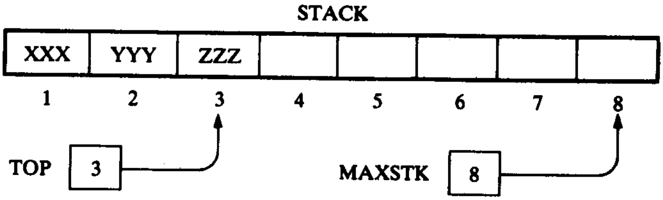
ازطرف دیگر فرض کنید توانایی کامل‌کردن پردازش پروژهٔ **D** را داریم. آنگاه تنها پروژه‌ای که می‌توانیم پردازش آن را ادامه دهیم پروژهٔ **C** است که در بالای پشته است. از این‌رو پوشهٔ پروژهٔ **C** را از پشته حذف می‌کنیم، پشته به صورتی که در نمودار ۴ـ ۶ (د) به تصویر کشیده شده است باقی می‌ماند و پردازش **C** ادامه می‌یابد. به همین ترتیب پس از کامل‌شدن پردازش **C**، پوشهٔ **B** را از پشته حذف می‌کنیم و پشته به صورتی که در نمودار ۴ـ ۶ (ه) به تصویر کشیده شده است باقی می‌ماند و پردازش **B** ادامه می‌یابد. بالاخره پس از کامل‌شدن پردازش **C**، آخرین پوشه، **A** را از پشته حذف می‌کنیم، پشتهٔ خالی باقی ماندهٔ در نمودار ۴ـ ۶ (و) به تصویر کشیده شده است و پردازش پروژهٔ اصلی ما **A** ادامه می‌یابد.

ملاحظه می‌کنید که در هر مرحله از پردازش بالا، پشته بطور اتوماتیک ترتیبی را نگه می‌دارید که نیازمند کامل‌کردن پردازش است. یک مثال مهم از چنین پردازشی در علوم کامپیوتر در جایی است که در آن **A** یک برنامهٔ اصلی است و **B**، **C** و **D** زیربرنامه‌هایی هستند که با ترتیبِ داده‌شده فراخوانده می‌شوند.

۳ـ ۶ نمایش پشته‌ها با آرایه

پشته‌ها را می‌توان درکامپیوتر به صورتهای مختلف، معمولاً به وسیلهٔ لیست یکطرفه یا آرایه خطی نمایش داد. هر یک از پشته‌های ما، به وسیلهٔ یک آرایهٔ خطی **STACK**، یک متغیر اشاره‌گر **TOP**، که حاوی مکان عنصر بالای پشته و یک متغیر **MAXSTK** است که بیشترین تعداد عناصر قابل نگهداری توسط پشته را به دست می‌دهد، نمایش داده می‌شود. اگر منظور ما غیر از این باشد به صورت صریح یا ضمنی بیان می‌کنیم. شرط **TOP = 0** یا **TOP = NULL** مبین آن است که پشته خالی است.

شکل ۵ـ ۶ چنین نمایشی از پشته را، توسط آرایه نشان می‌دهد. جهت سهولت در نمادگذاری، آرایه را به جای صورت عمودی آن، به صورت افقی رسم کرده‌ایم.



شکل ۵ـ ۶

از آنجاکه TOP = 3 ، پشته سه عنصر دارد، XXX و YYY و ZZZ ، چون MAXSTK = 8 ، جا برای 5 عنصر در پشته وجود دارد.

عمل اضافه‌کردن (Pushکردن) یک عنصر به درون یک پشته و عمل برداشتن یا حذف‌کردن (POP کردن) یک عنصر از یک پشته را می‌توان به ترتیب با زیربرنامه‌های Procedure زیر موسوم به PUSH و POP پیاده‌سازی کرد. در اجرای زیربرنامهٔ PUSH ، نخست باید تحقیق کنیم که آیا جا برای عنصر جدید در پشته وجود دارد یا خیر، اگر جواب منفی بود آنگاه وضعیت موسوم به سرریزی OverFlow را داریم. به‌طور مشابه، در اجرای زیربرنامهٔ POP نخست باید تحقیق کنیم که آیا عنصری در پشته برای حذف وجود دارد یا خیر، اگر جواب منفی است آنگاه وضعیت موسوم به زیرریزی UnderFlow را داریم.

Procedure 6.1: PUSH(STACK, TOP, MAXSTK, ITEM)
This procedure pushes an ITEM onto a stack.

1. [Stack already filled?]
   If TOP = MAXSTK, then: Print: OVERFLOW, and Return.
2. Set TOP := TOP + 1. [Increases TOP by 1.]
3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position.]
4. Return.

Procedure 6.2: POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. [Stack has an item to be removed?]
   If TOP = 0, then: Print: UNDERFLOW, and Return.
2. Set ITEM := STACK[TOP]. [Assigns TOP element to ITEM.]
3. Set TOP := TOP − 1. [Decreases TOP by 1.]
4. Return.

اغلب TOP و MAXSTK متغیرهای سراسری هستند از این‌رو زیربرنامه‌ها را می‌توان تنها با استفاده از

PUSH(STACK, ITEM) و POP(STACK, ITEM)

به ترتیب فرا خواند. خاطرنشان می‌کنیم که مقدار TOP قبل از اضافه‌شدن عنصر در PUSH تغییر می‌کند اما مقدار TOP بعد از حذف‌شدن عنصر در POP تغییر می‌کند.

مثال ۲ـ ۶

(الف) پشتهٔ شکل ۵ـ ۶ را درنظر بگیرید. عمل PUSH(STACK, WWW) را به صورت زیر شبیه‌سازی می‌کنیم :

1. Since TOP = 3, control is transferred to Step 2.
2. TOP = 3 + 1 = 4.
3. STACK[TOP] = STACK[4] = WWW.
4. Return.

توجه دارید که WWW اکنون عنصر بالای پشته است.

(ب) مجدداً پشتهٔ شکل ۵ـ ۶ را درنظر بگیرید. این‌بار عمل POP(STACK, ITEM) را به صورت زیر شبیه‌سازی می‌کنیم :

1. Since TOP = 3, control is transferred to Step 2.
2. ITEM = ZZZ.
3. TOP = 3 − 1 = 2.
4. Return.

ملاحظه می‌کنید که STACK[TOP] = STACK[2] = YYY اکنون عنصر بالای پشته است.

به حداقل رساندن سرریزی

یک تفاوت اساسی بین زیرریزی و سرریزی در ارتباط با پشته‌ها نمایان می‌شود. زیرریزی به میزان زیادی به الگوریتم داده شده و دادهٔ ورودی بستگی دارد و از این‌رو برنامه‌نویس هیچ کنترل مستقیمی بر آن ندارد. ازطرف دیگر، سرریزی بستگی به انتخاب برنامه‌نویس برای مقدار حافظه‌ای دارد که برای هر پشته ذخیره می‌کند، همچنین این انتخاب تعداد دفعات وقوع سرریزی را تحت‌الشعاع خود قرار می‌دهد.

در حالت کلی، تعداد عناصر یک پشته با اضافه‌شدن یا کم‌شدن عناصر تغییر می‌کند. بنابراین، انتخاب ویژه مقدار حافظه برای یک پشتهٔ داده شده، مستلزم توازن بین زمان و حافظه است. به‌ویژه این که، در ابتداء ذخیره مقدار زیاد حافظه برای هر پشته تعداد دفعات وقوع سرریزی را کاهش می‌دهد. با وجود این که در اکثر کارها به‌ندرت از حافظهٔ زیاد استفاده می‌شود، مصرف حافظه زیاد برای جلوگیری از

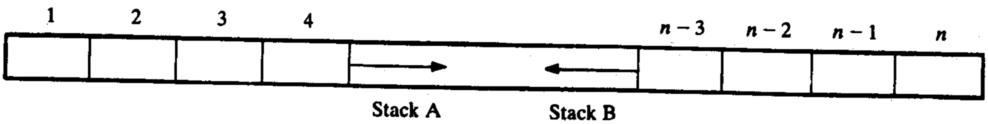
مسأله سرریزی پرهزینه خواهد بود و زمان مورد نیاز برای حل مسأله سرریزی، نظیر اضافه‌کردن حافظه به پشته می‌تواند پرهزینه‌تر از حافظهٔ ذخیره شده باشد.

روشهای متعددی وجود دارد که نمایش آرایه‌های پشته‌ها را به‌گونه‌ای اصلاح می‌کند که مقدار فضای ذخیره شده برای بیش از یک پشته را می‌تواند با کارآیی بیشتری مورد استفاده قرار دهد. اغلب این روشها خارج از حدود این درس است. یک نمونه از چنین روشی در مثال زیر بیان شده است.

مثال ۳ـ ۶

فرض کنید یک الگوریتم داده شده به دو پشتهٔ A و B احتیاج دارد. برای پشتهٔ A یک آرایه STACKA با n_1 عنصر و برای پشتهٔ B یک آرایه STACKB با n_2 عنصر می‌توان تعریف کرد. سرریزی وقتی اتفاق می‌افتد که یا پشتهٔ A شامل بیش از n_1 عنصر باشد یا پشتهٔ B بیش از n_2 عنصر داشته باشد.

فرض کنید بجای این که یک آرایه STACK با $n = n_1+n_2$ عنصر برای پشته‌های A و B تعریف کنیم نظیر آنچه که در شکل ۶ـ ۶ به تصویر درآمده است، STACK[1] را به صورت پائین پشتهٔ A تعریف کنیم و به A اجازه دهیم به طرف راست رشد کند و STACK[n] را به صورت پائین پشتهٔ B تعریف کنیم و به B اجازه دهیم به طرف چپ رشد کند. در این حالت، سرریزی تنها وقتی اتفاق می افتد که A و B بیش از $n = n_1 + n_2$ عنصر داشته باشند. این روش معمولاً تعداد دفعات وقوع سرریزی راکاهش می‌دهد حتی اگر ما تعدادِ کل فضای ذخیره شدهٔ برای دو پشته را افزایش ندهیم. در استفاده از این ساختمان داده عملیات PUSH و POP لازم است اصلاح شوند.



شکل ۶ـ ۶

۴ـ ۶ عبارتهای محاسباتی ؛ نمادگذاری لهستانی

فرض کنید Q یک عبارت محاسباتی شامل ثابت‌ها و عملیات ریاضی باشد. این بخش الگوریتمی را ارائه می‌دهد که مقدار Q را با استفاده از نمادگذاری لهستانی معکوس یا نمادگذاری پسوندی پیدا می‌کند. ملاحظه خواهید کرد که پشته ابزار اساسی برای این الگوریتم است.

یادآور می‌شویم که عملیات دودویی در Q ممکن است دارای سطوح تقدّم مختلف باشند. به‌ویژه این که فرض را بر سه سطح تقدم یا اولویت زیر برای پنج عمل دودویی متداول قرار می‌دهیم:

بالاترین اولویت · توان ↑

بعد از بالاترین اولویت : ضرب (*) و تقسیم (/)

پائین‌ترین اولویت : جمع (+) و تفریق (−)

ملاحظه می‌کنید که ما برای توان از نماد زبان BASIC استفاده می‌کنیم. جهت سهولت فرض می‌کنیم که Q شامل هیچ عمل یگانی نیست (نظیر علامت منفی در ابتدای عبارت b −). علاوه بر این فرض می‌کنیم که در تمام عبارت بدون پرانتز، عملیات هم‌سطح از چپ به راست اجرا می‌شوند. این فرض استاندارد نیست چون برخی از زبانهای برنامه‌نویسی عمل توان‌رساندن را از راست به چپ اجرا می‌کنند.

مثال ۴ـ ۶

فرض کنید بخواهیم عبارت محاسباتی بدون پرانتز زیر را ارزیابی کنیم :

$$2 \uparrow 3 + 5 * 2 \uparrow 2 - 12 / 6$$

نخست توان را ارزیابی می‌کنیم، نتیجه چنین است :

$$8 + 5 * 4 - 12 / 6$$

آنگاه ضرب و تقسیم را ارزیابی می‌کنیم که به‌دست می‌آید $8 + 20 - 2$ درنهایت جمع و تفریق را ارزیابی می‌کنیم که نتیجهٔ نهایی 26 است. ملاحظه می‌کنید که این عبارت سه بار پیمایش می شود که هربار متناظر با یک سطح از اولویت عملیات است.

نمادگذاری لهستانی

در متداول‌ترین عملیات محاسباتی، عملگر بین دو عملوند قرار می‌گیرد. به عنوان مثال

$$A + B \quad C - D \quad E * F \quad G / H$$

این نمادگذاری، **نمادگذاری میانوندی** نامیده می‌شود. با این نمادگذاری، مابین دو عبارت

$$(A + B) * C \quad \text{و} \quad A + (B * C)$$

با استفاده از پرانتزگذاری‌ها یا برخی از قراردادهای اولویت عملگرها نظیر سطوحِ متداول اولویت‌ها که در بالا مورد بررسی قرار گرفت تمایز قائل هستیم. بنابراین، ترتیب عملگرها و عملوندها در یک عبارت محاسباتی با توجه به ترتیبی که در آن عملیات اجرا می‌شوند به‌طور منحصربفرد تعیین نمی‌شود.

نمادگذاری لهستانی : این نامگذاری که پس از ریاضیدان لهستانی **یان لوکاسیویچ** صورت گرفته است مربوط به نمادگذاری‌ای می‌شود که در آن، عملگر قبل از دو عملوند قرار می‌گیرد.

برای مثال :

$$+AB \quad -CD \quad *EF \quad /GH$$

ما مرحله به مرحله، عبارتهای میانوندی زیر را با استفاده از دو کروشه [ ] به نماد لهستانی تبدیل می‌کنیم که مبین تبدیل قسمت به قسمت آن است :

$$(A+B)*C=[+AB]*C=*+ABC$$
$$A+(B*C)=A+[*BC]=+A*BC$$
$$(A+B)/(C-D)=[+AB]/[-CD]=/+AB-CD$$

خاصیت اساسی نمادگذاری لهستانی آن است، ترتیبی که در آن عملیات انجام می‌شوند به وسیلهٔ مکان عملگرها و عملوندهای عبارت به‌طور کامل تعیین می‌شود. بنابراین هنگام نوشتن عبارتها با نماد لهستانی به هیچ پرانتزی نیاز نیست.

نماد لهستانی معکوس درارتباط با نمادگذاری مشابه‌ای است که در آن نماد عملگر پس از دو عملوند قرار می‌گیرد :

AB + CD − EF * GH /

در اینجا نیز برای تعیین ترتیب عملیات هر عبارت محاسباتی نوشته شده با نماد لهستانی معکوس به هیچ پرانتزی نیاز نیست. این نمادگذاری اغلب **نمادگذاری پسوندی** نامیده می‌شود درحالی که **نمادگذاری پیشوندی** اصطلاحی است که برای **نمادگذاری لهستانی** بکار می‌رود که در پاراگراف قبل مورد بررسی قرار گرفت.

کامپیوتر معمولاً عبارت محاسباتی نوشته شده به صورت نمادگذاری میانوندی را در دو مرحله ارزیابی می‌کند. نخست عبارت را به صورت نمادگذاری پسوندی تبدیل می‌کند و آنگاه عبارت پسوندی را ارزیابی می‌کند. در هر مرحله، پشته، ابزار اصلی‌ای است که برای انجام این کار مشخص مورد استفاده قرار می‌گیرد. ما این کاربرد پشته‌ها را به‌ترتیب عکس نشان می‌دهیم، یعنی نخست نشان می‌دهیم که چگونه از پشته‌ها برای ارزیابی عبارت پسوندی استفاده می‌شود و آنگاه نشان می‌دهیم که چگونه از پشته‌ها در تبدیل عبارتهای میانوندی به صورت عبارتهای پسوندی استفاده می‌شود.

ارزیابی یک عبارت پسوندی

فرض کنید **P** یک عبارت محاسباتی نوشته شده با نماد پسوندی باشد. الگوریتم زیر که از یک **STACK** برای نگهداری عملوندها استفاده می‌کند **P** را ارزیابی می‌کند.

Algorithm 6.3: This algorithm finds the VALUE of an arithmetic expression P written in postfix notation.

1. Add a right parenthesis ")" at the end of P. [This acts as a sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel ")" is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator ⊗ is encountered, then:
   (*a*) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   (*b*) Evaluate B ⊗ A.
   (*c*) Place the result of (*b*) back on STACK.
   [End of If structure.]
   [End of Step 2 loop.]
5. Set VALUE equal to the top element on STACK.
6. Exit.

توجه دارید که هنگام اجرای مرحلهٔ 5 تنها یک عدد در STACK خواهد بود.

مثال ۵ـ ۶

عبارت محاسباتی P ی زیر راکه به صورت نماد پسوندی نوشته شده است درنظر بگیرید :

P: 5, 6, 2, +, *, 12, 4, /, −

از کاماها به این دلیل برای جداکردن عناصر P استفاده کرده‌ایم تا 5,6,2 به صورت عدد 562 تعبیر نشود.

Symbol Scanned		STACK
(1)	5	5
(2)	6	5, 6
(3)	2	5, 6, 2
(4)	+	5, 8
(5)	*	40
(6)	12	40, 12
(7)	4	40, 12, 4
(8)	/	40, 3
(9)	−	37
(10)	)	

شکل ۷ـ ۶

عبارت میانوندی Q معادل آن به صورت زیر است :

Q: $5 * (6 + 2) - 12 / 4$

توجه دارید که در عبارت میانوندی Q به پرانتز احتیاج داریم اما در عبارت پسوندی P هیچ احتیاجی به پرانتزگذاری نیست.

P را با شبیه‌سازی الگوریتم 6.3 ارزیابی می‌کنیم. نخست یک پرانتز بستهٔ نگهبان درانتهای P اضافه می‌کنیم که به دست می‌آید :

P:	5,	6,	2,	+,	*,	12,	4,	/,	−,	)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)

جهت سهولت در مراجعه عناصر P ، از چپ به راست شماره‌گذاری شده‌اند. شکل ۷ـ ۶ محتوای STACK را به‌محض جستجو و خواندن هر عنصر P نشان می‌دهد. عدد آخر در STACK یعنی 37 که در VALUE جایگزین شده است، هنگامی که نگهبان ")" را جستجو و می‌خواند مقدار P است.

تبدیل عبارتهای میانوندی به عبارتهای پسوندی

فرض کنید Q یک عبارت محاسباتی باشد که با نماد میانوندی نوشته شده است. علاوه بر عملوندها و عملگرها، Q نیز می‌تواند شامل پرانتزهای چپ و راست باشد. فرض می‌کنیم که عملگرها در Q تنها شامل عملگرهای توان (↑) ، ضرب (*)، تقسیم (/) ، جمع (+) و تفریق (−) می‌باشد و مانند بالا سه سطح تقدم یا اولویت متداول دارد. علاوه براین فرض می‌کنیم عملگرهای هم‌سطح، من‌جمله عملگرهای توان از چپ به راست اجرا می‌شوند مگر آن که با پرانتزگذاری خلاف آن بیان شود. این قرارداد استاندارد نیست چون عبارتها می‌توانند شامل عملگرهای یکانی باشند و برخی از زبانهای برنامه‌نویسی عمل توان‌رساندن را از راست به چپ اجرا می‌کنند. با وجود این، ما از این فرضها جهت ساده‌شدن الگوریتمها استفاده می‌کنیم.

الگوریتم زیر عبارت میانوندی Q را به عبارت پسوندی معادل آن P تبدیل می‌کند. این الگوریتم از یک پشته برای نگهداری موقت عملگرها و پرانتزهای چپ استفاده می‌کند. عبارت پسوندی P با استفاده از عملوندها از Q و عملگرهایی که از STACK حذف می‌شود از چپ به راست ساخته می‌شوند. کار را با Push کردن پرانتز چپ به درون STACK شروع می‌کنیم و در پایان Q یک پرانتز راست اضافه می‌کنیم. الگوریتم هنگامی کامل می‌شود که STACK خالی باشد.

Algorithm 6.4: POLISH(Q, P)
Suppose Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator ⊗ is encountered, then:
   (*a*) Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same precedence as or higher precedence than ⊗.
   (*b*) Add ⊗ to STACK.
   [End of If structure.]
6. If a right parenthesis is encountered, then:
   (*a*) Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   (*b*) Remove the left parenthesis. [Do not add the left parenthesis to P.]
   [End of If structure.]
   [End of Step 2 loop.]
7. Exit.

اصطلاحی که گاهی اوقات در مرحلهٔ 5 مورد استفاده قرار می‌گیرد⊗ است که به سطح پائین خودش اشاره می‌کند.

مثال ۶ـ ۶

عبارت محاسباتی میانوندی Q زیر را درنظر بگیرید :

$$Q: \quad A + (B * C - (D / E \uparrow F) * G) * H$$

الگوریتم 6.4 را شبیه‌سازی می‌کنیم تا Q را به عبارت پسوندی معادل آن P تبدیل کند. نخست "(" را به داخل STACK ، Push می‌کنیم و آنگاه ")" را درانتهای Q اضافه می‌کنیم که حاصل می‌شود :

Q:	A	+	(	B	*	C	−	(	D	/	E	↑	F	)	*	G	)	*	H	)
	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)	(15)	(16)	(17)	(18)	(19)	(20)

عنصرهای Q اکنون جهت سهولت در مراجعه به آنها از چپ به راست شماره‌گذاری شده‌اند. شکل ۶ـ۸ وضعیت STACK و وضعیت رشتهٔ P را وقتی که هر عنصر Q جستجو و خوانده می‌شود نشان می‌دهد. ملاحظه می‌کنید که

(۱) هر عملوند فقط به P اضافه می‌شود و STACKتغییر نمی‌کند.

(۲) عملگر تفریق (−) در ردیف 7، * را از STACK به P ارسال می‌کند قبل از آنها (−) به داخل STACK، Push شود.

(۳) پرانتز راست در ردیف 14، ↑ و آنگاه / را از STACK به P ارسال می‌کند و آنگاه پرانتز چپ را از بالای STACK حذف می‌کند.

(۴) پرانتز راست در ردیف 20، * و آنگاه + را از STACK به P ارسال می‌کند و آنگاه پرانتز چپ را از بالای STACK حذف می‌کند.

پس از اجرای مرحلهٔ 20، STACK خالی است و

P: A B C * D E F ↑ / G * − H * +

که عبارت پسوندی موردنیاز معادل Q است.

Symbol Scanned	STACK	Expression P
(1) A	(	A
(2) +	(+	A
(3) (	(+ (	A
(4) B	(+ (	A B
(5) *	(+ (*	A B
(6) C	(+ (*	A B C
(7) −	(+ (−	A B C *
(8) (	(+ (− (	A B C *
(9) D	(+ (− (	A B C * D
(10) /	(+ (− (/	A B C * D
(11) E	(+ (− (/	A B C * D E
(12) ↑	(+ (− (/ ↑	A B C * D E
(13) F	(+ (− (/ ↑	A B C * D E F
(14))	(+ (−	A B C * D E F ↑ /
(15) *	(+ (− *	A B C * D E F ↑ /
(16) G	(+ (− *	A B C * D E F ↑ / G
(17))	(+	A B C * D E F ↑ / G * −
(18) *	(+ *	A B C * D E F ↑ / G * −
(19) H	(+ *	A B C * D E F ↑ / G * − H
(20))		A B C * D E F ↑ / G * − H * +

شکل ۸ـ ۶

۵ـ ۶ QUICKSORT، یک کاربرد از پشته‌ها

فرض کنید A یک لیست با n عنصر داده‌ای باشد. منظور ما از "مرتب‌کردن A" عمل تجدید آرایش عناصر A است تا این عناصر با یک ترتیب منطقی کنار هم قرار گیرند. یعنی وقتی که A از داده‌های عددی تشکیل می‌شود به صورت عددی مرتب شده باشند و وقتی A از داده‌های کاراکتری تشکیل می‌شود به

صورت الفبایی مرتب شده باشند. موضوع مرتب‌کردن، به همراه الگوریتم‌های مختلف آن بطور اساسی در فصل ۹ مورد بررسی قرار می‌گیرد. این بخش تنها یک الگوریتم مرتب‌کردن را ارائه می‌دهد که الگوریتم **QuickSort** نام دارد تا یک کاربرد از پشته‌ها را نشان می‌دهد.

QuickSort الگوریتمی از نوعِ تقسیم و غلبه است. به بیان دیگر، مسألهٔ مرتب‌کردن یک مجموعه به مسأله مرتب‌کردن دو مجموعهٔ کوچکتر تبدیل می‌شود. ما این مرحلهٔ ساده‌شدن را به کمک یک مثال مشخص توضیح می‌دهیم.

فرض کنید **A** لیست **12** عددی زیر باشد :

44, 33, 11, 55, 77, 90, 40, 60, 99, 22, 88, 66

مرحلهٔ ساده‌شدن الگوریتم **QuickSort** مکان نهایی یکی از اعداد را پیدا می‌کند. دراین مثال از اولین عدد یعنی **44** استفاده می‌کنیم. این کار به صورت زیر انجام می‌شود. با آخرین عدد یعنی **66** شروع می‌کنیم لیست را از راست به چپ پیمایش می‌کنیم هر عدد را با **44** مقایسه می‌کنیم و کار را در نخستین عدد کوچکتر از **44** متوقف می‌کنیم. این عدد **22** است. جای **44** و **22** را عوض می‌کنیم، لیست زیر به دست می‌آید :

22, 33, 11, 55, 77, 90, 40, 60, 99, 44, 88, 66

(ملاحظه می‌کنید که اعداد **88** و **66** که در طرف راست **44** هستند بزرگتر از **44** می‌باشند.) با **22** شروع می‌کنیم به دنبال آن لیست را در جهت مخالف، از چپ به راست پیمایش می‌کنیم. هر دو عدد را با **44** مقایسه کرده و کار را در نخستین عدد بزرگتر از **44** متوقف می‌کنیم. این عدد **55** است. جای دو عدد **44** و **55** را عوض می‌کنیم،لیست زیر به‌دست می‌آید.

22, 33, 11, 44, 77, 90, 40, 60, 99, 55, 88, 66

(ملاحظه می‌کنید که اعداد **22** ، **33** و **11** در طرف چپ **44** همگی کوچکتر از **44** هستند) این بار با **55** شروع می‌کنیم، اکنون لیست را درجهت اصلی، از راست به چپ پیمایش می‌کنیم این کار را تا آنجا ادامه می‌دهیم تا به اولین عدد کوچکتر از **44** برسیم. این عدد **40** است. جای دو عدد **44** و **40** را عوض می‌کنیم، درنتیجهٔ آن، لیست زیر حاصل می‌شود :

22, 33, 11, 40, 77, 90, 44, 60, 99, 55, 88, 66

(مجدداً اعداد طرف راست **44** همگی بزرگتر از **44** هستند.) . با **40** شروع می‌کنیم، لیست را از چپ به راست پیمایش می‌کنیم. اولین عدد بزرگتر از **44** عدد **77** است. جای دو عدد **44** و **77** را عوض می‌کنیم، درنتیجهٔ آن، لیست زیر حاصل می‌شود :

22, 33, 11, 40, 44, 90, 77, 60, 99, 55, 88, 66

(این بار عدد طرف چپ 44 همگی کوچکتر از 44 هستند.) با 77 شروع می‌کنیم. لیست را از راست به چپ برای جستجوی یک عدد کوچکتر از 44 پیمایش می‌کنیم. قبل از ملاقات با 44 چنین عددی را ملاقات نمی‌کنیم. معنی آن این است که تمام اعداد پیمایش شدند و با 44 مقایسه شدند. علاوه بر این، تمام اعداد کوچکتر از 44 که اکنون در طرف چپ 44 هستند تشکیل لیست کوچکی از اعداد می‌دهند و تمام اعداد بزرگتر از 44 که اکنون در طرف راست 44 هستند تشکیل لیست کوچکی از اعداد می‌دهند. این دو وضعیت در زیر نشان داده شده است.

$$\underbrace{22,\quad 33,\quad 11,\quad 40,}_{\text{First sublist}}\quad 44,\quad \underbrace{90,\quad 77,\quad 60,\quad 99,\quad 55,\quad 88,\quad 66}_{\text{Second sublist}}$$

لیست طرف چپ   لیست طرف راست

بدین ترتیب 44 به صورت صحیحی در مکان نهایی‌اش قرار گرفته است و کار مرتب‌کردن لیست اصلی A اکنون به کار مرتب‌کردن هر یک از دو لیست طرف چپ و راست بالا ساده می‌شود.

مرحله ساده‌شدن بالا بر روی هر یک از دو لیست اخیر که شامل 2 یا چند عنصر است تکرار می‌شود. از آنجا که تنها می‌توانیم، یکی از دو زیرلیست را پردازش کنیم، باید توانایی نگهداری چند لیست کوچک را، برای پردازش آتی داشته باشیم. این کار با استفاده از دو پشته تحت نامهای **LOWER** و **UPPER** انجام می‌شود که بطور موقت چنین لیستهای کوچکی را نگه می‌دارد. به بیان دیگر آدرسهای اولین و آخرین عناصر هر یک از این لیستهای کوچک که **مقادیر کرانه‌ای** یا **مرزی** آن نامیده می‌شود به درون پشته‌های **LOWER** و **UPPER**، **PUSH** می‌شود و مرحلهٔ ساده‌سازی یک لیست کوچک تنها پس از حذف مقادیر مرزی آن از پشته‌ها، اعمال می‌شود. مثال زیر روش استفاده از پشته‌های **LOWER** و **UPPER** را روشن می‌سازد.

مثال ۷ـ ۶

لیست A بالا با $n = 12$ عنصر را درنظر بگیرید. الگوریتم با **PUSH**کردن مقادیر مرزی 1 و 12 از A به داخل پشته‌ها شروع می‌شود که به دست می‌آید :

LOWER: 1   UPPER: 12

برای اِعمال مرحله ساده‌سازی، نخست الگوریتم مقادیر 1 و 12 را از بالای پشته‌ها حذف می‌کند، باقی می‌ماند :

LOWER: (empty)   UPPER: (empty)

و آنگاه مرحله ساده‌شدن را به همان صورتی که در بالا انجام شد بر لیست متناظر **A[1], A[2], ..., A[12]** اعمال می‌کنیم. نهایتاً عنصر اول، یعنی 44 در **A[5]** قرار می‌گیرد. بنابراین الگوریتم مقادیر مرزی 1 و 4 از

لیست کوچک شدهٔ اول و مقادیر مرزی 6 و 12 از لیست کوچک‌شدهٔ دوم را به داخل پشته‌ها **Push** می‌کند، نتیجه می‌شود :

LOWER: 1, 6 UPPER: 4, 12

مجدداً برای اعمال مرحله ساده شدن، الگوریتم مقادیر 6 و 12 را از بالای پشته‌ها حذف می‌کند، نتیجه می‌شود :

LOWER: 1 UPPER: 4

و آنگاه مرحلهٔ ساده‌شدن را بر لیست کوچک متناظر آن **A[6], A[7], A[12],...,A[12]**, اعمال می‌کنیم. مرحله ساده‌شدن، این لیست را مطابق شکل ۹ـ ۶ تغییر می‌دهد.

A[6],	A[7],	A[8],	A[9],	A[10],	A[11],	A[12],
90,	77,	60,	99,	55,	88,	66
66,	77,	60,	99,	55,	88,	90
66,	77,	60,	90,	55,	88,	99
66,	77,	60,	88,	55,	90,	99

لیست کوچک اول **لیست کوچک دوم**

شکل ۹ـ ۶

ملاحظه می‌کنید که لیست کوچک دوم تنها یک عنصر دارد. بنابراین، الگوریتم تنها مقادیر مرزی 6 و 10 لیست کوچک اول را به داخل پشته‌ها **Push** می‌کند. نتیجه می‌دهد:

LOWER: 1, 6 UPPER: 4, 10

و الی آخر. الگوریتم وقتی پایان می‌یابد که پشته‌ها حاوی هیچ لیست کوچک‌شده‌ای که پردازش نشده است توسط مرحله ساده‌شدن نباشد.

بیان رسمی الگوریتم **QuickSort** به صورت زیر است. جهت سهولت در نمادگذاری و ملاحظات آموزشی، الگوریتم به دو قسمت تقسیم می‌شود. قسمت اول زیربرنامهٔ **Procedure** را ادامه می‌دهد که **QUICK** نام دارد و مرحلهٔ ساده‌سازی بالا را در الگوریتم انجام می‌دهد و قسمت دوم از **QUICK** برای مرتب‌کردن تمام لیست استفاده می‌کند.

ملاحظه می‌کنید که مرحلهٔ **(iii)** از **2(c)** ضروری نیست و جهت تأکید در تقارن بین مرحلهٔ **2** و مرحلهٔ **3** اضافه شده است. در این **Procedure** فرض نشده است که عناصر **A** از هم متمایز هستند. در غیر اینصورت شرط **LOC ≠ RIGHT** در مرحلهٔ **2(a)** و شرط **LEFT ≠ LOC** در مرحلهٔ **3(a)** قابل حذف

خواهد بود.

Procedure 6.5: QUICK(A, N, BEG, END, LOC)

Here A is an array with N elements. Parameters BEG and END contain the boundary values of the sublist of A to which this procedure applies. LOC keeps track of the position of the first element A[BEG] of the sublist during the procedure. The local variables LEFT and RIGHT will contain the boundary values of the list of elements that have not been scanned.

```
1. [Initialize.] Set LEFT := BEG, RIGHT := END and LOC := BEG.
2. [Scan from right to left.]
      (a) Repeat while A[LOC] ≤ A[RIGHT] and LOC ≠ RIGHT:
              RIGHT := RIGHT − 1.
          [End of loop.]
      (b) If LOC = RIGHT, then: Return.
      (c) If A[LOC] > A[RIGHT], then:
              (i) [Interchange A[LOC] and A[RIGHT].]
                      TEMP := A[LOC], A[LOC] := A[RIGHT],
                      A[RIGHT] := TEMP.
             (ii) Set LOC := RIGHT.
            (iii) Go to Step 3.
          [End of If structure.]
3. [Scan from left to right.]
      (a) Repeat while A[LEFT] ≤ A[LOC] and LEFT ≠ LOC:
              LEFT := LEFT + 1
          [End of loop.]
      (b) If LOC = LEFT, then: Return.
      (c) If A[LEFT] > A[LOC], then
              (i) [Interchange A[LEFT] and A[LOC].]
                      TEMP := A[LOC], A[LOC] := A[LEFT],
                      A[LEFT] := TEMP.
             (ii) Set LOC := LEFT.
            (iii) Go to Step 2.
          [End of If structure.]
```

الگوریتم قسمت دوم به صورت زیر است، همانگونه که در بالا ملاحظه کردید، **LOWER** و **UPPER** پشته‌هایی هستند که در آنها مقادیر مرزی لیستهای کوچک شده ذخیره می‌شود. طبق معمول از **NULL = 0** استفاده می‌کنیم.

```
Algorithm 6.6: (Quicksort) This algorithm sorts an array A with N elements.
    1. [Initialize.] TOP := NULL.
    2. [Push boundary values of A onto stacks when A has 2 or more elements.]
       If N > 1, then: TOP := TOP + 1, LOWER[1] := 1, UPPER[1] := N.
    3. Repeat Steps 4 to 7 while TOP ≠ NULL.
    4.     [Pop sublist from stacks.]
           Set BEG := LOWER[TOP], END := UPPER[TOP],
           TOP := TOP − 1.
    5.     Call QUICK(A, N, BEG, END, LOC). [Procedure 6.5.]
    6.     [Push left sublist onto stacks when it has 2 or more elements.]
           If BEG < LOC − 1, then:
               TOP := TOP + 1, LOWER[TOP] := BEG,
               UPPER[TOP] = LOC − 1.
           [End of If structure.]
    7.     [Push right sublist onto stacks when it has 2 or more elements.]
           If LOC + 1 < END, then:
               TOP := TOP + 1, LOWER[TOP] := LOC + 1,
               UPPER[TOP] := END.
           [End of If structure.]
       [End of Step 3 loop.]
    8. Exit.
```

پیچیدگی الگوریتم QuickSort

زمان اجرای یک الگوریتم مرتب‌کردن معمولاً با تعداد $f(n)$ دفعات مقایسهٔ موردنیاز برای مرتب‌کردن n عنصر اندازه‌گیری می‌شود. الگوریتم **QuickSort** که دارای تعداد : ادی مقایسه است به‌شدت مورد مطالعه و بررسی قرار گرفته است. درحالت کلی، این الگوریتم در بدترین حالت زمان اجرایی از مرتبهٔ $n^2 / 2$ دارد اما زمان اجرای حالت میانگین آن از مرتبهٔ $n\log n$ اس... ن آن در زیر ارائه شده است.

بدترین حالت وقتی اتفاق می‌افتد که لیست از قبل مرتب شده باشد. آنگاه نخستین عنصر به n مقایسه احتیاج دارد تا معلوم شود در مکان اول قرار می‌گیرد. علاوه بر این، لیست کوچک شدهٔ اول خالی خواهد بود اما لیست کوچک شدهٔ دوم $n - 1$ عنصر دارد. بنابراین عنصر دوم به $n - 1$ مقایسه احتیاج دارد تا معلوم شود در مکان دوم قرار می‌گیرد و الی آخر. درنتیجه، مجموعاً تعداد

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

مقایسه انجام می‌شود. ملاحظه می‌کنید که این عدد برابر پیچیدگی الگوریتم مرتب‌کردن حبابی است (ر. ک. بخش ۶-۴).

پیچیدگی $f(n) = O(n \log n)$ حالت میانگین از این واقعیت ناشی می‌شود که به‌طور متوسط، هر مرحلهٔ ساده‌سازی در الگوریتم دو لیست کوچکتر تولید می‌کند. بنابراین :

(۱) با ساده‌شدن لیست اولیه، 1 عنصر در جای خود قرار می‌گیرد و دو لیست کوچکتر تولید می‌شود.

(۲) با ساده‌شدن دو لیست، 2 عنصر در جای خود قرار می‌گیرند و چهار لیست کوچکتر تولید می‌شود.

(۳) با ساده‌شدن چهار لیست، 4 عنصر در جای خود قرار می‌گیرند و هشت لیست کوچکتر تولید می‌شود.

(۴) با ساده‌شدن دو لیست، 8 عنصر در جای خود قرار می‌گیرند و شانزده لیست کوچکتر تولید می‌شود.

و الی آخر. ملاحظه می‌کنید که مرحله ساده‌شدن در K امین سطح مکان عنصر 2^{k-1} ام را پیدا می‌کند. از این‌رو تقریباً $\log_2 n$ سطح ساده‌سازی وجود دارد. علاوه براین، هر سطح حداکثر از n مقایسه استفاده می‌کند. بنابراین $f(n) = O(n \log n)$ درواقع تحلیل ریاضی و ملاحظات تجربی هردو نشان می‌دهند که

$$f(n) \approx 1.4[n \log n]$$

تعداد انتظاری مقایسه‌ها برای الگوریتم QuickSort است.

۶ـ۶ زیربرنامه‌های بازگشتی

بازگشتی یک مفهوم بسیار مهم در علم کامپیوتر است. بسیاری از الگوریتم‌ها را می‌توان با استفاده از مفهوم بازگشتی به صورت کاراتری بیان کرد. این بخش، این ابزار قدرتمند را معرفی می‌کند و بخش ۸ـ۶ چگونگی پیاده‌سازی بازگشتی را با استفاده از پشته‌ها نشان می‌دهد.

فرض کنید P یک زیربرنامهٔ Procedure باشد که حاوی یک دستور Call است که خودش را صدا می‌کند یا حاوی یک دستور Call است که زیربرنامهٔ دوم را فرا می‌خواند که نهایتاً نتیجه دستور Call این است که زیربرنامهٔ Procedure اصلی را صدا می‌زند. در آن صورت به P یک **زیربرنامهٔ بازگشتی** می‌گویند، طوری که این برنامه به تعداد مرحلهٔ معین و محدودی اجرا می‌شود و پس از آن اجرا ادامه نمی‌یابد. هر زیربرنامهٔ بازگشتی باید دو خاصیت زیر را داشته باشد :

(۱) باید معیار معینی وجود داشته باشد که **معیار پایه یا مبنا** نام دارد و باتوجه به آن معیار، زیربرنامهٔ Procedure خودش را صدا نمی‌زند.

(۲) هرباری که زیربرنامهٔ Procedure (به‌طور مستقیم یا غیرمستقیم) خودش را صدا می‌زند، باید به معیار پایه نزدیکتر شود.

یک **زیربرنامهٔ بازگشتی** یا Recursive را با دو معیار بالا، **خوش تعریف** می‌گویند.

به‌طور مشابه، یک تابع را به صورت **بازگشتی تعریف شده** می‌گویند هرگاه تعریف تابع به خودش برگردد. مجدداً برای این که تعریف چرخشی نباشد، باید دارای دو خاصیت زیر باشد :

(۱) باید آرگومانهای مشخصی وجود داشته باشد که به آن **مقادیر پایه** می‌گویند که به‌ازای این مقادیر

تابع خودش را صدا نمی‌زند.

(۲) هربار که تابع خودش را صدا می‌زند، آرگومان تابع باید به مقدار پایه نزدیکتر شود.

یک تابع بازگشتی را با این دو خاصیت نیز **خوش‌تعریف** می‌گویند.

مثال‌های زیر در روشن‌شدن مفهوم بازگشتی به شما کمک خواهد کرد:

تابع فاکتوریل!

حاصل ضرب اعداد صحیح مثبت از 1 تا خود n ، n فاکتوریل نامیده می‌شود و معمولاً آن را با !n نمایش می‌دهند. بنابراین

$$n! = 1.2.3...(n-2)(n-1)n$$

بنابه قرارداد $0! = 1$ تعریف می‌شود. بدین ترتیب تابع فاکتوریل برای تمام اعداد صحیح مثبت تعریف می‌شود. بنابراین داریم

$$0! = 1 \qquad 1! = 1 \qquad 2! = 1\cdot 2 = 2 \qquad 3! = 1\cdot 2\cdot 3 = 6 \qquad 4! = 1\cdot 2\cdot 3\cdot 4 = 24$$

$$5! = 1\cdot 2\cdot 3\cdot 4\cdot 5 = 120 \qquad 6! = 1\cdot 2\cdot 3\cdot 4\cdot 5\cdot 6 = 720$$

و الی آخر، ملاحظه می‌کنید که

$$5! = 5.4! = 5.24 = 120 \quad \text{و} \quad 6! = 6.5! = 6.120 = 720$$

یعنی برای هر عدد صحیح مثبت n تساوی زیر برقرار است :

$$n! = n.(n-1)!$$

بنابراین تابع فاکتوریل را می‌توان به صورت زیر نیز تعریف کرد :

تعریف ۱ـ ۶ : (تابع فاکتوریل)

(الف) اگر $n = 0$، آنگاه $n! = 1$.

(ب) اگر $n>0$، آنگاه $n! = n.(n-1)!$.

ملاحظه می‌کنید که این تعریف !n بازگشتی است، چون وقتی از $(n-1)$ استفاده می‌کند به خودش مراجعه می‌کند. بنابراین (الف) مقدار !n به صورت صریح داده می‌شود حتی وقتی $n = 0$ است بدین ترتیب 0 مقدار پایه است و (ب) مقدار !n به‌ازای n دلخواه برحسب مقدار کوچکتر n تعریف می‌شود که به مقدار پایهٔ 0 نزدیک است. بنابراین، تعریف چرخشی نیست یا به عبارت دیگر، این تابع خوش تعریف است.

مثال ۶ـ ۶

با استفاده از تعریف بازگشتی فاکتوریل، !4 را محاسبه کنید. این محاسبه نیازمند نه مرحلهٔ زیر است:

(1) $4! = 4 \cdot 3!$
(2) $3! = 3 \cdot 2!$
(3) $2! = 2 \cdot 1!$
(4) $1! = 1 \cdot 0!$
(5) $0! = 1$
(6) $1! = 1 \cdot 1 = 1$
(7) $2! = 2 \cdot 1 = 2$
(8) $3! = 3 \cdot 2 = 6$
(9) $4! = 4 \cdot 6 = 24$

یعنی:

مرحلهٔ ۱: در این مرحله !4 برحسب !3 تعریف می‌شود، از این‌رو تا هنگام ارزیابی !3 باید محاسبهٔ !4 به تعویق بیفتد. این تعویق با نوشتن مرحله بعدی به صورت پله‌ای مشخص شده است.

مرحلهٔ ۲: در اینجا !3 برحسب !2 تعریف می‌شود، از این‌رو تا هنگام ارزیابی !2 باید محاسبهٔ !3 به تعویق بیفتد.

مرحلهٔ ۳: این مرحله !2 را برحسب !1 تعریف می‌کند.

مرحلهٔ ۴: این مرحله !1 را برحسب !0 تعریف می‌کند.

مرحلهٔ ۵: این مرحله !0 را می‌تواند به صورت صریح ارزیابی کند چون 0 مقدار پایهٔ تعریف بازگشتی است.

مرحله‌های ۶ تا ۹. حال از آخر به اول برمی‌گردیم. برای محاسبهٔ !1 از !0 استفاده می‌کنیم، از !1 برای محاسبهٔ !2، از !2 برای محاسبهٔ !3 و بالاخره از !3 برای محاسبهٔ !4 استفاده می‌کنیم. این برگشت از آخر به اول با نوشتن مراحل محاسبات به صورت پله‌ای معکوس شده، بیان شده است.

ملاحظه می‌کنید که ما به‌ترتیب عکس از محاسبات به تعویق افتاده اصلی به اول برگشتیم. یادآور می‌شویم این نوع از پردازش به تعویق افتاده خود منتهی به استفاده از پشته‌ها می‌شود. بخش ۲ـ ۶ را ببینید.

در زیر دو زیربرنامهٔ Procedure ارائه شده است که هر یک از آنها n فاکتوریل را محاسبه می‌کند:

Procedure 6.7A: FACTORIAL(FACT, N)
This procedure calculates N! and returns the value in the variable FACT.

1. If N = 0, then: Set FACT := 1, and Return.
2. Set FACT := 1. [Initializes FACT for loop.]
3. Repeat for K = 1 to N.
   Set FACT := K * FACT.
   [End of loop.]
4. Return.

Procedure 6.7B: FACTORIAL(FACT, N)
This procedure calculates N! and returns the value in the variable FACT.

1. If N = 0, then: Set FACT := 1, and Return.
2. Call FACTORIAL(FACT, N − 1).
3. Set FACT := N * FACT.
4. Return.

ملاحظه می‌کنید که زیربرنامهٔ اول با استفاده از پردازش حلقه‌های تکرار N! را ارزیابی می‌کند. ازطرف دیگر، زیربرنامهٔ دوم یک زیربرنامهٔ بازگشتی است چون دارای پردازشی است که خودش را صدا می‌کند. برخی از زبانهای برنامه‌نویسی، به‌ویژه FORTRAN استفاده از چنین زیربرنامه‌های بازگشتی را مجاز نمی‌دانند.

فرض کنید P یک زیربرنامهٔ بازگشتی باشد. درطی اجرای یک الگوریتم یا یک برنامه که حاوی P است، به هربار اجرای زیربرنامهٔ P به صورت زیر یک **شمارهٔ سطح** نسبت می‌دهیم. در اجرای زیربرنامهٔ اصلی P سطح 1 جایگزین می‌شود و هربار که زیربرنامهٔ P اجرا می‌شود، بخاطر یک احضار بازگشتی، سطح آن 1 واحد بیشتر از سطح اجرایی است، که باعث احضار بازگشتی شده است. در مثال ۸ـ ۶ ، **مرحلهٔ ۱**، سطح ۱ دارد. از این‌رو مرحلهٔ ۲ سطح ۲ ، مرحلهٔ ۳ سطح ۳ ، مرحلهٔ ۴ سطح ۴ و مرحلهٔ ۵ سطح ۵ دارد. از سوی دیگر، مرحلهٔ ۶ سطح ۴ دارد چون نتیجهٔ یک بازگشت از سطح ۵ است. به بیان دیگر، مرحلهٔ ۶ و مرحلهٔ ۴ متعلق به همان سطح اجرا هستند. بطور مشابه، مرحلهٔ ۷ سطح ۳ ، مرحلهٔ ۸ سطح ۲ و مرحلهٔ نهایی، مرحلهٔ ۹ ، سطح اولیهٔ ۱ دارد.

عمق بازگشتیِ یک زیربرنامهٔ بازگشتی P با یک مجموعهٔ معین از آرگومانها، بزرگترین شماره سطح P در طول اجرای آن است.

دنبالهٔ فیبوناچی

دنبالهٔ زیبا و معروف فیبوناچی که معمولاً با F_0 ، F_1 ، F_2 ، ... نمایش داده می‌شود به صورت زیر است :

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, \ldots$$

یعنی $F_0 = 0$ و $F_1 = 1$ و هر جملهٔ بعدی مجموع دو جملهٔ قبلی است. برای مثال، دو جملهٔ بعدی دنبالهٔ بالا به صورت زیر محاسبه می‌شود :

$$34 + 55 = 89 \quad \text{و} \quad 55 + 89 = 144$$

تعریف رسمی این تابع به صورت زیر است :

تعریف ۲ـ ۶ : (دنباله فیبوناچی)

(الف) اگر $n = 0$ یا $n = 1$، آنگاه $F_n = n$.

(ب) اگر $n>1$، آنگاه $F_n = F_{n-2} + F_{n-1}$.

این مثال دیگری از یک تعریف بازگشتی است، چون وقتی از F_{n-2} و F_{n-1} استفاده می‌کند این تعریف به خودش برمی‌گردد. در اینجا در (الف) مقادیر پایه 0 و 1 هستند و در (ب) مقدار F_n برحسب مقادیر کوچکتر از n تعریف می‌شود که نزدیک به مقادیر پایه هستند. بنابراین این تابع خوش تعریف است.

زیربرنامهٔ Procedure ای که جمله n ام دنبالهٔ فیبوناچی F_n را پیدا می‌کند به صورت زیر است :

Procedure 6.8: FIBONACCI(FIB, N)

این زیربرنامه F_N را محاسبه می‌کند و مقدار آن را در پارامتر اول FIB قرار می‌دهد و به برنامهٔ احضارکننده تحویل می‌دهد :

1. If N = 0 or N = 1, then: Set FIB := N, and Return.
2. Call FIBONACCI(FIBA, N − 2).
3. Call FIBONACCI(FIBB, N − 1).
4. Set FIB := FIBA + FIBB.
5. Return.

این مثال دیگری از یک زیربرنامهٔ بازگشتی است، چون زیربرنامهٔ Procedure خودش را احضار می‌کنند. درواقع امر، این زیربرنامه دوبار خودش را احضار می‌کند. متذکر می‌شویم (ر.ک. مسأله ۱۶ـ ۶) که می‌توان یک زیربرنامهٔ Procedure با روش تکرار برای محاسبهٔ F_n نوشت که در آن از زیربرنامهٔ بازگشتی استفاده نشود.

الگوریتم‌های تجزیه یا الگوریتم‌های تقسیم و غلبه

مسأله P را که در ارتباط با مجموعهٔ S است درنظر بگیرید. فرض کنید A الگوریتمی است که S را به مجموعه‌های کوچکتر افراز می‌کند به‌گونه‌ای که حل مسأله P برای S، به حل P برای یک یا چند مجموعهٔ کوچکتر منتهی شود. آنگاه A یک **الگوریتم تقسیم و غلبه** نامیده می‌شود.

دو مثال از الگوریتم‌های تقسیم و غلبه که قبلاً مورد بررسی قرار گرفت الگوریتمها QuickSort در بخش ۵ـ ۶ و الگوریتم جستجوی دودویی در بخش ۷ـ ۴ است، یادآوری می‌کنیم که الگوریتم QuickSort از یک مرحله ساده‌شدن برای تعیین مکان یک عنصر و مسأله مرتب‌کردن تمام مجموعه برای مسأله مرتب‌کردن مجموعه‌های کوچکتر استفاده می‌کند. الگوریتم جستجوی دودویی مجموعهٔ

مرتبِ داده شده را به دو نیمه تقسیم می‌کند به‌گونه‌ای که مسأله جستجو برای یک عنصر در تمام مجموعه به مسأله جستجوی آن عنصر در یکی از دو نیمه منتهی می‌شود.

الگوریتم تقسیم و غلبهٔ A را می‌توان به صورت یک زیربرنامهٔ بازگشتی مورد توجه قرار داد. به این دلیل که وقتی آن را بر مجموعه‌های کوچکتر اعمال می‌کنیم الگوریتم A خودش را احضار می‌کند. معیار پایه و اصلی،این الگوریتم‌ها معمولاً مجموعه‌های یک عنصری است. برای مثال در الگوریتم مرتب‌کردن، مجموعهٔ یک عنصری بطور خودکار مرتب شده است و در الگوریتم جستجو، مجموعهٔ یک عنصری تنها نیازمند یک مقایسه است.

تابع آکرمان Ackermann

تابع آکرمان یک تابع با دو آرگومان است که در هر یک از این آرگومانها می‌تواند هر عدد صحیح غیرمنفی : 0 , 1 , 2 ,... جایگزین شود.

این تابع به صورت زیر تعریف می‌شود :

تعریف ۳ـ ۶ : (تابع آکرمان)

(الف) اگر $m = 0$ آنگاه $A(m, n) = n + 1$.

(ب) اگر $m \neq 0$ اما $n = 0$ آنگاه $A(m, n) = A(n - 1, 1)$.

(ج) اگر $m \neq 0$ و $n \neq 0$ آنگاه $A(m, n) = A(m - 1, A(m, n - 1))$.

در تابع آکرمان یک بار بیشتر، تعریف بازگشتی داریم چون تعریف قسمت‌های(ب)و(ج) به خودش رجوع می‌کند.

ملاحظه می‌کنید که $A(m,n)$ به صورت صریح تنها وقتی $m = 0$ است داده شده است. معیار پایه زوجهای مرتب زیر هستند :

$$(0, 0), \quad (0, 1), \quad (0, 2), \quad (0, 3),\ldots,(0, n),\ldots$$

هرچند از تعریف روشن نمی‌شوداما مقدار هر $A(m, n)$ را می‌توان در نهایت برحسب مقدار تابع و براساس یک یا چند زوج مرتب پایه بیان کرد.

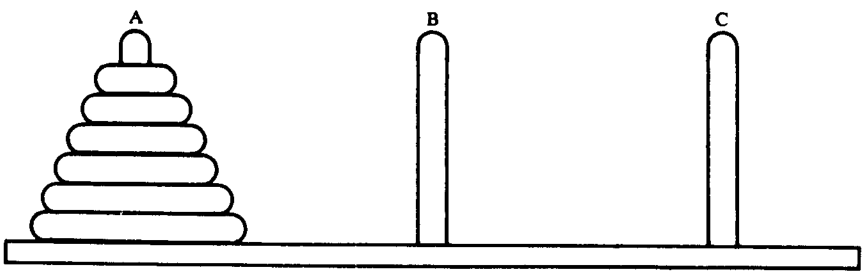
مقدار $A(1,3)$ در مسأله ۱۷ـ ۶ محاسبه می‌شود. حتی این حالت ساده نیازمند ۱۵ مرحله است. در حالت کلی، تابع آکرمان آنقدر پیچیده است که هر مقدار دلخواه را نمی‌توان با آن محاسبه کرد اما امکان محاسبهٔ مثالهای بدیهی و ساده وجود دارد. اهمیت تابع آکرمان از کاربرد آن در منطق ریاضی ناشی

می‌شود. در اینجا این تابع اساساً به این خاطر بیان شده است تا مثال دیگری از یک تابع بازگشتی کلاسیک را ارائه دهد و نشان می‌دهیم که قسمت بازگشتی این تعریف ممکن است پیچیده باشد.

۷ـ ۶ برجهای هانوی

در بخش قبل مثالهایی چند از تعریفهای بازگشتی و زیربرنامه‌های بازگشتی ارائه دادیم. این بخش چگونگی استفاده از زیربرنامهٔ بازگشتی را به عنوان ابزاری جهت توسعهٔ یک الگوریتم نشان می‌دهد که یک مسأله خاص را حل می‌کند. مسأله‌ای که برای این کار انتخاب شده است به مسأله برج‌های هانوی معروف است.

سه میله را درنظر بگیرید که با برچسب A ، B ، C مشخص شده‌اند و فرض کنید روی میلهٔ A تعداد معین، n دیسک با اندازه‌های مختلف از بزرگ به کوچک قرار داده شده است. برای حالت $n = 6$ این وضعیت در شکل ۱۰ـ ۶ نشان داده شده است.



شکل ۱۰ـ ۶ . وضعیت اولیهٔ ایجاد برجهای هانوی با $n = 6$ دیسک

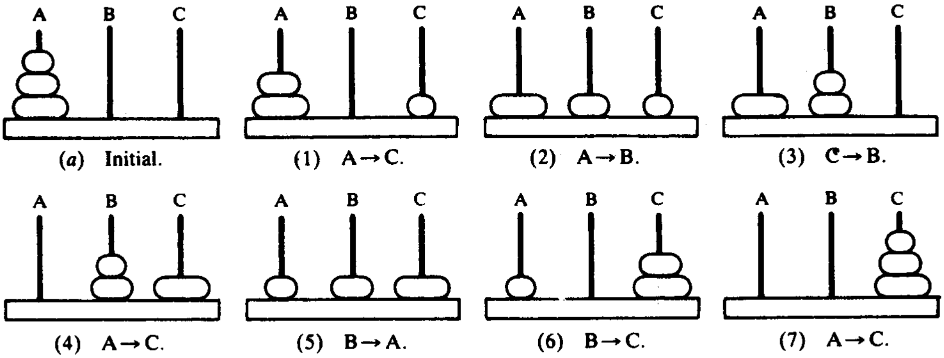
هدف بازی برج هانوی آن است که دیسکها را با استفاده از میلهٔ کمکی B ، از میلهٔ A به میلهٔ C منتقل کنیم، قوانین این بازی به صورت زیر است :

(الف) در هر بار تنها یک دیسک را می‌توان انتقال داد. به‌طور مشخص، تنها دیسک بالایی هر میله را می‌توان به هر میلهٔ دیگر منتقل کرد.

(ب) در هیچ زمانی نمی‌توان دیسک بزرگتر را روی دیسک کوچکتر قرار داد.

گاهی اوقات نمایش دستور "دیسک بالای میلهٔ X را به میلهٔ Y منتقل کنید" را به صورت $X \rightarrow Y$ می‌نویسیم که در آن X و Y می‌تواند هر یک از سه میلهٔ داده شده باشد.

در شکل ۱۱ـ۶ راه حل مسأله برجهای هانوی برای n = 3 دیسک ارائه شده است.



شکل ۱۱ـ۶

ملاحظه می‌کنید که حل مسأله برجهای هانوی در این حالت، از هفت انتقال یا جابجایی زیر تشکیل شده است:

n = 3 : دیسک بالای میلهٔ A را به میلهٔ C منتقل کنید.
دیسک بالای میلهٔ A را به میلهٔ B منتقل کنید.
دیسک بالای میلهٔ C را به میلهٔ B منتقل کنید.
دیسک بالای میلهٔ A را به میلهٔ C منتقل کنید.
دیسک بالای میلهٔ B را به میلهٔ A منتقل کنید.
دیسک بالای میلهٔ B را به میلهٔ C منتقل کنید.
دیسک بالای میلهٔ A را به میلهٔ C منتقل کنید.

به بیان دیگر،

$n=3$: A→C, A→B, C→B, A→C, B→A, B→C, A→C

برای روشن شدن مطلب، حل مسأله برجهای هانوی را برای n = 1 و n = 2 نیز می‌نویسیم:

$n=1$: A→C
$n=2$: A→B, A→C, B→C

توجه دارید که در n = 1 تنها از یک جابجایی یا انتقال دیسک استفاده می‌کند و برای n = 2 از سه انتقال استفاده شده است.

به عوض پیداکردن یک راه حل مجزا برای هر n دلخواه، ما از روش بازگشتی برای ابداع یک راه حل کلی استفاده می‌کنیم. نخست ملاحظه می‌کنید که حل مسأله برجهای هانوی برای n>1 دیسک می‌تواند

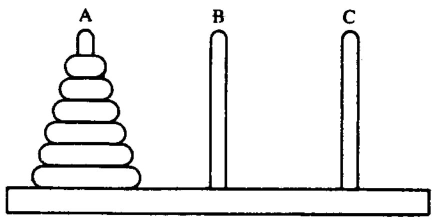
منجر به سه زیر مسأله داده شده در زیر می‌شود :

(۱) تعداد 1 – n دیسک بالا را از میلهٔ A به میلهٔ B منتقل کنید.

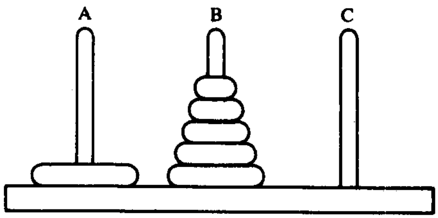
(۲) دیسک بالا از میلهٔ A را به میلهٔ C منتقل کنید: $A \longrightarrow C$.

(۳) تعداد 1 – n دیسک بالا را از میلهٔ B به میلهٔ C منتقل کنید.

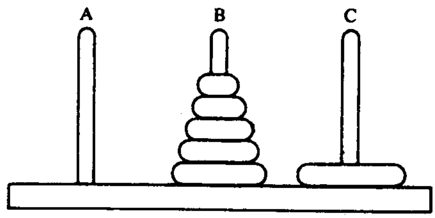
به ازای 6 = n این مسأله‌های کاهش مراحل در شکل ۱۲-۶ نشان داده شده است یعنی نخست پنج دیسک بالای میلهٔ A را به میلهٔ B منتقل می‌کنیم، آنگاه دیسک بزرگ را از میلهٔ A به میلهٔ C منتقل می‌کنیم و بدنبال آن پنج دیسک بالای میلهٔ B را به میلهٔ Cمنتقل می‌کنیم:



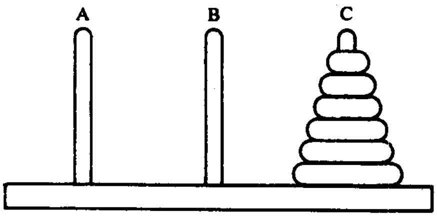
(الف) وضعیت اولیه:



(ب) انتقال پنج دیسک بالای میلهٔ A به میلهٔ B



(ج) انتقال دیسک بالای میلهٔ A به میلهٔ C



(د) انتقال پنج دیسک بالای میلهٔ B به میلهٔ C

شکل ۱۲-۶

اکنون وقت آن رسیده است که نماد اصلی را معرفی کنیم:

TOWER(N, BEG, AUX, END)

نماد بالا زیربرنامهٔ Procedure ای را نشان می‌دهد که با استفاده از میلهٔ کمکی AUX ، n دیسک بالای وضعیت اولیهٔ میلهٔ BEG را به وضعیت نهایی میلهٔ END منتقل می‌کند. وقتی 1 = n است راه حل بدیهی زیر را داریم :

TOWER(1, BEG, AUX, END) تنها از دستور $BEG \longrightarrow END$ تشکیل می‌شود. علاوه بر این همانگونه که در بالا مورد بررسی قرار گرفت وقتی n>1 ، حل آن به حل سه زیرمسأله داده شده در زیر منتهی می‌شود :

TOWER(N − 1, BEG, END, AUX) (۱)

TOWER(1, BEG, AUX, END) or BEG→END (۲)

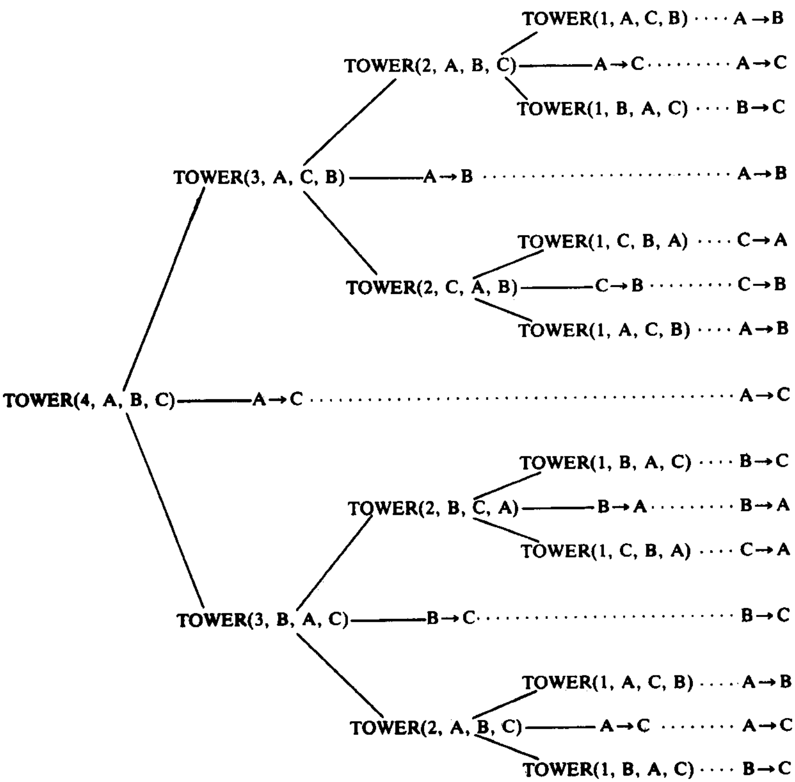
TOWER(N − 1, AUX, BEG, END) (۳)

ملاحظه می‌کنید که هر یک از این سه زیرمسأله را می‌توان مستقیماً حل کرد یا اساساً همانند مسأله اصلی است با این تفاوت که در آن از تعداد دیسکهای کمتری استفاده شده است. بنابراین، پردازش این مسأله‌های کاهش مراحل، منتهی به یک راه حل بازگشتی برای مسأله برجهای هانوی می‌شود.

شکل ۱۳ـ ۶ نمودارِ حلِ بازگشتی مسأله بالا برای

TOWER(4, A, B, C)

است.



شکل ۱۳ـ ۶ حل بازگشتی مسأله برجهای هانوی برای n = 4 دیسک

ملاحظه می‌کنید که حل بازگشتی مسأله برجهای هانوی برای $n = 4$ دیسک شامل 15 انتقال یا جابجایی زیر است :

A→B A→C B→C A→B C→A C→B A→B A→C
B→C B→A C→A B→C A→B A→C B→C

در حالت کلی، این حل بازگشتی برای n دیسک مستلزم $f(n) = 2^n-1$ جابجایی یا انتقال است.

بررسی خود را روی برجهای هانوی با زیربرنامهٔ Procedure که به صورت رسمی نوشته شده است خلاصه می‌کنیم :

Procedure 6.9: TOWER(N, BEG, AUX, END)
This procedure gives a recursive solution to the Towers of Hanoi problem for N disks.

1. If N = 1, then:
   (*a*) Write: BEG→END.
   (*b*) Return.
   [End of If structure.]
2. [Move N − 1 disks from peg BEG to peg AUX.]
   Call TOWER(N − 1, BEG, END, AUX).
3. Write: BEG→END.
4. [Move N − 1 disks from peg AUX to peg END.]
   Call TOWER(N − 1, AUX, BEG, END).
5. Return.

این زیربرنامه یک راه حل بازگشتی برای مسأله برجهای هانوی برای n دیسک ارائه می‌دهد.

می‌توان این راه حل را به صورت یک الگوریتم تقسیم و غلبه مورد توجه قرار داد، چون راه حل مسأله برای n دیسک به راه حلی برای $n - 1$ و راه حلی برای $n=1$ دیسک منجر می‌شود.

۸ـ ۶ پیاده‌سازی زیربرنامه‌های بازگشتی به وسیلهٔ پشته‌ها

در بخشهای قبل نشان داده‌ایم که چگونه برنامه‌های بازگشتی می‌تواند برای مسائل خاص یک ابزار مفید در توسعه الگوریتم‌ها باشد. در این بخش چگونگی استفاده از پشته‌ها در پیاده سازی زیربرنامه‌های بازگشتی نشان داده می‌شود. بهتر است ابتدا زیربرنامه‌ها را در حالت کلی مورد بحث و بررسی قرار دهیم.

یادآوری می‌کنیم که یک زیربرنامه می‌تواند شامل همهٔ پارامترها و همه متغیرهای محلی باشد. پارامترها، متغیرهایی هستند که مقادیر را از متغیرهای برنامه فراخواننده موسوم به **آرگومانها** دریافت می‌کنند و سپس مقادیر را به برنامه فراخواننده انتقال می‌دهند. علاوه بر پارامترها و متغیرهای محلی، زیربرنامه نیز باید آدرس بازگشت برنامهٔ فراخواننده را نگهدارند. این آدرس بازگشتی اساسی و دارای

اهمیت است چون کنترل کار باید به مکان واقعی‌اش در برنامهٔ فراخواننده منتقل شود. زمانی که اجرای زیربرنامه به پایان می‌رسد و کنترل کار به برنامهٔ فراخواننده داده می‌شود به مقادیر متغیرهای محلی و آدرس بازگشتی، دیگر نیازی نیست.

فرض کنید زیربرنامهٔ ما یک زیربرنامهٔ بازگشتی است. آنگاه هر سطح اجرای زیربرنامه می‌تواند شامل مقادیر مختلف برای پارامترها و متغیرهای محلی و آدرس بازگشتی باشد. علاوه براین، اگر برنامهٔ بازگشتی خودش را احضار کند، آنگاه این مقادیر جاری و گذرا، باید نگهداری شوند، چون هنگامی که برنامه مجدداً فعال می‌شود از آنها استفاده خواهد کرد.

فرض کنید یک برنامه‌نویس از یک زبان سطح بالا نظیر PASCAL استفاده می‌کند که اجازهٔ استفادهٔ از زیربرنامه‌های بازگشتی را می‌دهد. آنگاه کامپیوتر از حافظه‌هایی برای نگهداری تمام مقادیر پارامترها، متغیرهای محلی و آدرسهای بازگشتی استفاده می‌کند. ازطرف دیگر، اگر یک برنامه‌نویس از یک زبان برنامه‌نویسی سطح بالای دیگری نظیر FORTRAN استفاده کند که اجازهٔ استفاده از زیربرنامه‌های بازگشتی را نمی‌دهد آنگاه برنامه‌نویس باید تمهیداتی به عمل آورد تا زیربرنامهٔ بازگشتی را به یک زیربرنامهٔ غیربازگشتی تبدیل کند. این تمهیدات در زیر توضیح داده می‌شود.

تبدیل یک زیربرنامهٔ بازگشتی به یک زیربرنامهٔ غیربازگشتی

فرض کنید P یک زیربرنامهٔ بازگشتی باشد. فرض می‌کنیم P به عوض یک زیربرنامهٔ تابعی، یک زیربرنامهٔ Subroutine باشد. (بدون این که عمومیت مسأله از دست برود، زیرا زیربرنامه‌های تابع را می‌توان به‌سادگی به صورت زیربرنامه‌های Subroutine نوشت.) علاوه بر این فرض می‌کنیم که فقط زیربرنامهٔ P است که خود P را به صورت بازگشتی فرا می‌خواند. بررسی بازگشتی به صورت غیرمستقیم خارج از حدود مطالب درس ساختمان داده‌ها و این کتاب است.

در تبدیلِ زیربرنامهٔ بازگشتی P به یک زیربرنامهٔ غیربازگشتی به صورت زیر عمل می‌کنیم. قبل از هر چیز تعریفهای زیر را داریم:

(۱) برای هر پارامتر STPAR یک پشتهٔ PAR داریم.

(۲) برای هر متغیر محلی STVAR یک پشتهٔ VAR داریم.

(۳) برای نگهداری آدرسهای بازگشتی یک متغیر محلی ADD و یک پشتهٔ STADD داریم.

هر باری که یک احضار بازگشتی به P وجود دارد، مقادیر جاری پارامترها و متغیرهای محلی برای پردازش آتی به داخل پشته های مربوطه Push می‌شوند و هر باری که یک بازگشت بازگشتی به P وجود