به نام خدا

ساختمان داده

آرش شفیعی

داده ساختارهای پایه

مقدمه

- در این بخش با چند داده ساختار پایه از جمله آرایه‌ها [1] ، ماتریس‌ها [2] ، پشته‌ها [3] ، صف‌ها [4] ، و لیست‌های پیوندی [5] آشنا خواهیم شد.

---

[1] arrays
[2] matrices
[3] stacks
[4] queues
[5] linked lists

آرایه‌ها

- یک آرایه داده ساختاری است که دنباله‌ای از عناصر (که هرکدام مقداری را نگه‌داری می‌کند) را در حافظه ذخیره می‌کند. هرکدام از عناصر آرایه با یک اندیس تعیین می‌شوند. اندیس در واقع مکان یک عنصر در آرایه را مشخص می‌کند.

- اگر اندیس اول آرایه s باشد و آرایه در آدرس حافظه a ذخیره شود و هرکدام از عناصر آرایه b بایت را در حافظه اشغال کنند، آنگاه i امین عنصر آرایه در حافظه در بایت $a + b(i - s)$ تا $a + b(i - s + 1) - 1$ قرار می‌گیرد.

- اگر آرایه با اندیس ۱ شروع شود، عنصر i ام بایت‌های $a + b(i - 1)$ تا $a + bi - 1$ را اشغال می‌کند. اگر آرایه با اندیس ۰ آغاز شود، آنگاه عنصر i ام آرایه بایت‌های $a + bi$ تا $a + b(i + 1) - 1$ را اشغال می‌کند.

- با فرض بر اینکه کامپیوتر می‌تواند به همهٔ فضاهای حافظه مستقیما در یک زمان معین دسترسی پیدا کند، دسترسی به عناصر آرایه در زمان ثابت صورت می‌گیرد.

آرایه‌ها

- درج: اگر بخواهیم در انتهای یک آرایه عنصری را درج کنیم، کافی است مقدار عنصر جدید را در آخرین خانه آرایه قرار دهیم و این کار در زمان $O(1)$ انجام می‌شود. اما اگر بخواهیم عنصری جدید را در ابتدای آرایه درج کنیم باید هر یک از عناصر آرایه را یک خانه به جلو انتقال دهیم که این کار در زمان $O(n)$ در بدترین حالت برای آرایه‌ای با n عنصر انجام می‌شود. همچنین اگر بخواهیم عنصری را در مکانی دلخواه در آرایه درج کنیم، در بدترین حالت به زمان $O(n)$ نیاز داریم.

- حذف: اگر بخواهیم عنصری را از آرایه حذف کنیم، در بدترین حالت به زمان $O(n)$ نیاز داریم، زیرا عناصر بعد از عنصر حذف شده باید هر کدام یک خانه به سمت ابتدای آرایه انتقال داده شوند.

- جستجو: برای جستجوی یک مقدار در یک آرایه در بدترین حالت به زمان $O(n)$ نیاز داریم.

جستجوی دودویی

- برای جستجوی یک مقدار در یک آرایه باید همهٔ عناصر آرایه را یک‌به‌یک بررسی کنیم. این جستجو برای یک آرایه با n عنصر در زمان $O(n)$ انجام می‌شود.
- حال فرض می‌کنیم می‌خواهیم یک مقدار را در یک آرایه مرتب شده پیدا کنیم.
- برای این کار می‌توانیم از الگوریتمی به نام جستجوی دودویی [1] استفاده کنیم.

[1] binary search

جستجوی دودویی

- الگوریتم جستجوی دودویی آرایه را به دو قسمت تقسیم می‌کند. برای جستجوی مقدار x در آرایه A ، ابتدا مقدار x با عنصر وسط آرایه یعنی A[n/2] مقایسه می‌شود. اگر x برابر با مقدار وسط آرایه بود، مقدار مورد نظر یافته شده است. اگر x کوچکتر از عنصر وسط آرایه بود، باید x را در نیمه اول آرایه یعنی A[1:n/2-1] جستجو کنیم. در غیراینصورت باید x را در نیمه دوم آرایه یعنی A[n/2+1:n] جستجو کنیم. این روند را برای زیر آرایه‌ها ادامه می‌دهیم تا یا x یافته شود یا مشخص شود که x در آرایه وجود ندارد.

جستجوی دودویی

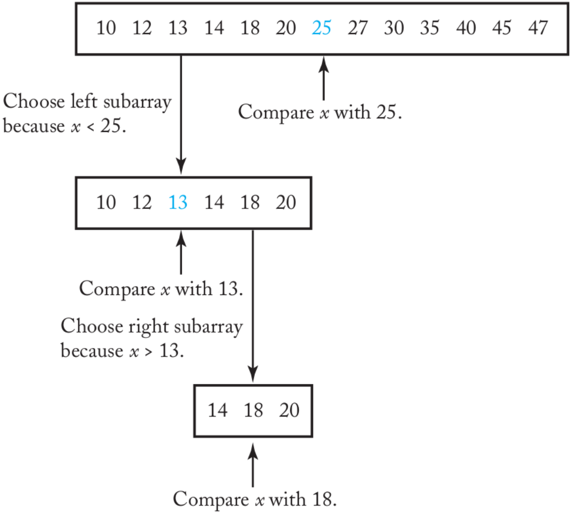
- بنابراین مراحل انجام جستجوی دودویی به صورت زیر است.

۱. برای پیدا کردن مقدار `x` در آرایه `A[low:high]` قرار می‌دهیم `mid=⌊(low + high)/2⌋` . اگر `A[mid]` برابر با `x` بود به نتیجه رسیده‌ایم در غیراینصورت آرایه را به دو قسمت `A[low:mid-1]` و `A[mid+1:high]` تقسیم می‌کنیم. این تقسیم تنها در صورتی می‌تواند انجام شود که `high` از `low` بزرگ‌تر باشد.

۲. در صورتی که مقدار `x` از `A[mid]` کوچکتر بود، الگوریتم جستجو برای `A[low:mid-1]` فراخوانی می‌شود، در غیراینصورت برای `A[mid+1:high]` فراخوانی می‌شود.

جستجوی دودویی

- برای پیدا کردن عدد ۱۸ در آرایهٔ زیر، الگوریتم به صورت زیر عمل می‌کند.



جستجوی دودویی

- الگوریتم جستجوی دودویی به صورت زیر است.

Algorithm Binary Search

```
function BinarySearch(A, x, low, high)
if (low > high) then
    return -1
mid = ⌊(low + high)/2⌋
if (x == A[mid]) then
    return mid
if (x < A[mid]) then
    return BinarySearch (A, x, low, mid-1)
else
    return BinarySearch (A, x, mid+1, high)
```

- برای جستجوی مقدار x جستجوی دودویی باید به صورت `BinarySearch(A, x, 1, n)` فراخوانی شود.

جستجوی دودویی

- در تقسیم یک آرایه به دو قسمت صرفا یک عملیات تقسیم در زمان $O(1)$ انجام می‌شود.
- بنابراین زمان اجرای الگوریتم جستجوی دودویی برای آرایه با n عنصر برابر است با زمان اجرای الگوریتم برای آرایه‌ای با $n/2$ عنصر به علاوه یک زمان ثابت.
- می‌توانیم بنویسیم $T(n) = T(\frac{n}{2}) + O(1)$ و $T(1) = O(1)$.
- با حل این رابطه بازگشتی به دست می‌آوریم $T(n) = O(\lg n)$.

ماتریس‌ها

- ماتریس یک آرایه دو بعدی است که می‌توانیم آن را توسط چند آرایه یک بعدی نمایش دهیم.
- دو روش معمول ذخیره ماتریس‌ها ترتیب سطری [1] و ترتیب ستونی [2] نام دارند.
- فرض کنید یک ماتریس با ابعاد $\mathrm{m} \times \mathrm{n}$ یا به عبارت دیگر یک ماتریس با m سطر و n ستون داریم.
- در ترتیب سطری، ماتریس سطر به سطر در حافظه ذخیره می‌شود و در ترتیب ستونی، ماتریس ستون به ستون ذخیره می‌شود.

---
[1] row-major order
[2] column-major order

ماتریس‌ها

- برای مثال ماتریس $M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ با ابعاد 3×2 را در نظر بگیرید.
- در ترتیب سطری ماتریس به صورت $1,\ 2,\ 3\ ;\ 4,\ 5,\ 6$ در حافظه ذخیره می‌شود و در ترتیب ستونی ماتریس به صورت $1,\ 4\ ;\ 2,\ 5\ ;\ 3,\ 6$ در حافظه ذخیره می‌شود.
- در شکل زیر نشان داده شده است که این ماتریس چگونه در یک آرایه در ترتیب سطری و ترتیب ستونی ذخیره می‌شود.

1	4	2	5	3	6

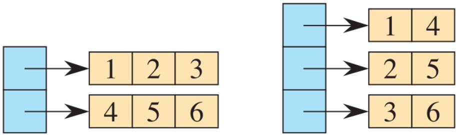
1	2	3	4	5	6

ماتریس‌ها

- بنابراین عنصر $M[i, j]$ در ترتیب سطری در اندیس $n(i-s)+j$ قرار می‌گیرد و در ترتیب ستونی در اندیس $m(j-s)+i$ قرار می‌گیرد.
- وقتی $s = 1$ است، در ترتیب سطری اندیس عنصر $M[i, j]$ برابر است با $n(i-1)+j$ و در ترتیب ستونی برابر است با $i + m(j-1)$.
- وقتی $s = 0$ است، در ترتیب سطری اندیس عنصر $M[i, j]$ برابر است با $ni + j$ و در ترتیب ستونی برابر است با $mj + i$.
- برای مثال عنصر $M[2, 1]$ در ماتریسی با ابعاد 3×2 ، وقتی $s = 1$ است، با ترتیب سطری در مکان $3(2-1)+1 = 4$ ذخیره می‌شود و با ترتیب ستونی در مکان $2 + 2(1-1) = 2$ ذخیره می‌شود.

ماتریس‌ها

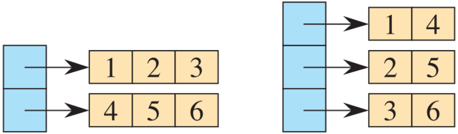
- همچنین ماتریس را می‌توان با استفاده از چند آرایه ذخیره کرد. در ترتیب سطری هر سطر در یک آرایه مجزا ذخیره می‌شود و در ترتیب ستونی هر ستون در یک آرایه مجزا ذخیره می‌شود.
- در شکل زیر یک ماتریس در دو ترتیب سطری و ستونی با استفاده از چند آرایه ذخیره شده است.



ماتریس‌ها

- در ترتیب سطری هر سطر در یک آرایه n عنصری ذخیره می‌شود. یک آرایه دیگر حاوی m عنصر است که در شکل به رنگ آبی نشان داده شده است. هریک از عناصر این آرایه به یکی از سطرهای آرایه اشاره می‌کند. فرض کنید آرایه آبی رنگ را A بنامیم. آنگاه $A[i]$ به سطر i ام ماتریس M اشاره می‌کند و عنصر $A[i][j]$ عنصر $M[i,j]$ را ذخیره می‌کند.

- در ترتیب ستونی، هر ستون در یک آرایه ذخیره می‌شود. تعداد n آرایه در این حالت وجود دارد که اندازه هر کدام m است. عنصر $M[i,j]$ در عنصر $A[j][i]$ ذخیره می‌شود.



ماتریس‌ها

- نمایش تک آرایه‌ای ماتریس‌ها کارایی بالاتری دارد. مزیت نمایش چند آرایه‌ای این است که می‌تواند ماتریس‌هایی را ذخیره کند که اندازه سطرها و ستون‌های آنها متفاوت است و بنابراین انعطاف پذیری بالاتری دارد.

ماتریس‌ها

- اگر درایه‌های یک ماتریس اکثراً برابر با صفر باشند، به آن ماتریس یک ماتریس خلوت [1] گفته می‌شود.
- در مقابل، اگر درایه‌های یک ماتریس اکثراً غیر صفر باشند، به آن ماتریس یک ماتریس چگال [1] یا متراکم گفته می‌شود.
- یک ماتریس خلوت را می‌توان به صورت چند آرایه‌ای ذخیره کرد، بدین صورت که یک آرایه برای سطرها در نظر گرفته، و در آرایه متناظر با هر سطر تنها درایه‌های غیرصفر را با ذکر شماره ستون آنها درج کنیم.

[1] sparse matrix
[1] dense matrix

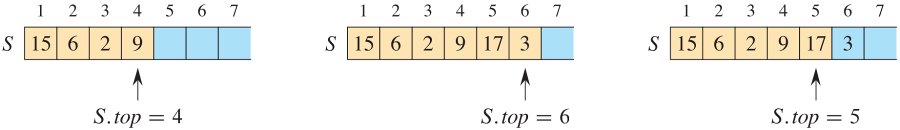
پشته

- پشته [1] داده ساختاری است که در آن امکان درج و حذف عناصر وجود دارد، به طوری که وقتی عملیات حذف بر روی پشته اعمال می‌شود، آخرین عنصری که به پشته اضافه شده است، حذف می‌شود.
- پشته بر اساس استراتژی خروج به ترتیب عکس ورودی پیاده‌سازی می‌شود بدین معنی که اولین عنصری که وارد پشته می‌شود آخر از همه از پشته خارج می‌شود. این استراتژی LIFO [2] نامیده می‌شود.
- عملیات درج در پشته Push و عملیات حذف از پشته Pop نامیده می‌شوند.
- در زبان انگلیسی به عملیات برداشتن یک ظرف از روی پشته‌ای از ظروف Pop و به عملیات گذاشتن یک ظرف بر روی پشته‌ای از ظروف Push گفته می‌شود و بدین دلیل این اسامی در ساختار داده پشته استفاده شده‌اند.
- ترتیب برداشتن ظروف از روی پشته‌ای از ظروف برعکس ترتیب قرار دادن آن‌ها بر روی پشته است.

[1] stack
[2] last-int first-out

پشته

- شکل زیر چگونگی پیاده‌سازی پشته‌ای از n عنصر را در آرایه $S[1:n]$ نشان می‌دهد.



- پشته یک ویژگی S.top دارد که اندیس آخرین عنصری است که به پشته اضافه شده است. ویژگی S.size اندازه یا ظرفیت پشته را مشخص می‌کند که همان اندازهٔ آرایه‌ای است که پشته با استفاده از آن پیاده‌سازی شده است. عناصر پشته در $S[1:S.top]$ قرار می‌گیرند. عنصر $S[1]$ عنصر انتهای [1] پشته و عنصر $S[S.top]$ عنصر روی [2] پشته نامیده می‌شوند.

[1] bottom
[2] top

پشته

- وقتی $S.top = 0$ است، پشته هیچ عنصری را شامل نمی‌شود و خالی [1] است.
- تابعی به نام Stack-Empty بررسی می‌کند آیا پشته خالی است یا خیر.
- اگر بخواهیم از یک پشته خالی عنصری برداریم با خطای پشته خالی [2] مواجه می‌شویم. همچنین اگر S.top بیشتر از S.size شود با خطای سرریز پشته [3] مواجه می‌شویم.

[1] empty
[2] underflow
[3] overflow

پشته

- تابع Stack-Empty در زیر پیاده‌سازی شده است. پیچیدگی زمانی این تابع $O(1)$ است.

Algorithm Stack Empty

```
function STACK-EMPTY(S)
if S.top = 0 then
    return true
else
    return false
```

پشته

- تابع Push در زیر پیاده‌سازی شده است. پیچیدگی زمانی این تابع $O(1)$ است.

Algorithm Push

```
function Push(S,x)
if S.top == S.size then
    error "overflow"
else
    S.top = S.top + 1
    S[S.top] = x
```

پشته

- تابع Pop در زیر پیاده‌سازی شده است. پیچیدگی زمانی این تابع $O(1)$ است.

Algorithm Pop

```
function Pop(S)
if Stack-Empty(S) then
    error "underflow"
else
    S.top = S.top - 1
    return S[S.top + 1]
```

- پشته کاربردهای زیادی در طراحی الگوریتم‌ها دارد که یک مثال از آنها را در اینجا بررسی می‌کنیم.
- یک عبارت ریاضی در نشانه‌گذاری پسوندی [1] عبارتی است که در آن عملگر بعد از عملوندها قرار می‌گیرد.
- در نشانه‌گذاری معمول که نشانه‌گذاری میانوندی [1] نامیده می‌شود، یک عملگر بین دو عملگر قرار می‌گیرد.
- برای مثال عبارت میانوندی $3 + 2$ در نشانه‌گذاری پسوندی به صورت $+ \; 3 \; 2$ نوشته می‌شود.

[1] postfix notation
[1] infix notation

پشته

- عبارت میانوندی $(1+2)-4$ در نشانه‌گذاری پسوندی به صورت $4\ 2\ 1\ +\ -$ و عبارت میانوندی $1+(2-4)$ در نشانه‌گذاری پسوندی به صورت $4\ 2\ -\ 1\ +$ نوشته می‌شود.
- همچنین عبارت میانوندی $4*3+2$ در نشانه‌گذاری پسوندی به صورت $2\ 3\ 4\ *\ +$ و عبارت میانوندی $4+3*2$ در نشانه‌گذاری پسوندی به صورت $2\ 3\ *\ 4\ +$ نوشته می‌شود.
- یکی از مزایای مهم نشانه‌گذاری پسوندی این است که برای محاسبه عبارت‌های پسوندی به پرانتزگذاری و بررسی اولویت عملگرها نیاز نیست.

پشته

- عبارات پسوندی را می‌توان با استفاده از یک پشته محاسبه کرد.
- الگوریتم محاسبه یک عبارت پسوندی به صورت زیر است.
- به ازای هر عملوندی که از ورودی خوانده می‌شود، مقدار آن در یک پشته ذخیره می‌شود. به ازای هر عملگری که از ورودی خوانده می‌شود، دو عملوند از پشته برداشته می‌شود، مقدار آنها با استفاده از عملگر خوانده شده محاسبه می‌شود، در نهایت مقدار به دست آمده در پشته ذخیره می‌شود. این عملیات ادامه می‌یابد تا اینکه ورودی کاملا خوانده شود. اگر تنها یک مقدار در پشته باقی بماند، آن مقدار نتیجه عبارت ورودی است، و اگر پشته خالی بماند یا بیشتر از یک مقدار داشته باشد، ورودی عبارتی نادرست بوده است.
- تمرین اول: تابعی بنویسید که با استفاده از یک پشته، یک عبارت میانوندی را به یک عبارت پسوندی تبدیل کند.
- تمرین دوم: تابعی بنویسید که با استفاده از یک پشته، مقدار یک عبارت پسوندی را محاسبه کند.

صف

- صف [1] داده ساختاری است که در آن عناصر به همان ترتیبی که وارد می‌شوند از آن خارج می‌شوند. به عبارت دیگر اولین عنصر وارد شده در صف اولین عنصری است که از آن خارج می‌شود.
- صف استراتژی FIFO [2] را پیاده‌سازی می‌کند بدین معنا که اولین عنصر وارد شده اولین عنصری است که خارج می‌شود.
- عملیات درج در صف Enqueue و عملیات حذف Dequeue نامیده می‌شوند.
- داده‌ساختار صف دقیقا همانند صف‌هایی است که در مکان‌های عمومی برای خدمت‌رسانی ایجاد می‌شود. اولین مشتری که وارد صف می‌شود اولین کسی است که از صف خارج شده و خدمت‌رسانی می‌شود.

[1] queue
[2] first-in first-out

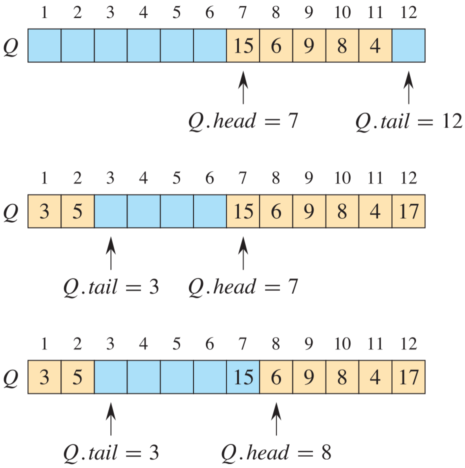
صف

- یک صف شامل یک ابتدا [1] و یک انتها [2] است.
- وقتی یک عنصر وارد صف می‌شود در انتهای صف قرار می‌گیرد همانند وقتی که یک مشتری وارد صف می‌شود. عنصری که از صف خارج می‌شود نیز عنصر ابتدای صف است، همانند اولین مشتری در صف که به او خدمت رسانی می‌شود.

[1] head
[2] tail

صف

- شکل زیر روشی برای پیاده‌سازی صفی را نشان می‌دهد که $n - 1$ عنصر دارد. این صف توسط آرایه $Q[1:n]$ پیاده‌سازی شده است.



صف

- ویژگی $Q.size$ اندازه صف است که برابر با طول آرایه (n) است. صف یک ویژگی به نام $Q.head$ دارد که اندیسی است که به ابتدای صف اشاره می‌کند. ویژگی $Q.tail$ اندیسی است که به مکان بعد از آخرین عنصر صف اشاره می‌کند. عناصر صف در مکان‌های $Q.head$ ، $Q.head+1$ ، تا $Q.tail-1$ قرار می‌گیرند.

- وقتی $Q.head = Q.tail$ است، صف خالی است. در ابتدا داریم $Q.head = Q.tail = 1$. در این حالت، اگر بخواهیم از صف عنصری خارج کنیم با خطای صف خالی [1] مواجه می‌شویم.

- وقتی $Q.head = Q.tail + 1$ یا $Q.head = 1$ و $Q.tail = Q.size$ می‌گوییم صف پر است. در این حالت اگر بخواهیم عنصری وارد صف کنیم با خطای سر ریز صف [2] مواجه می‌شویم.

---
[1] queue underflow
[2] queue overflow

صف

- در زیر تابع Enqueue پیاده سازی شده است.

Algorithm Enqueue

```
function ENQUEUE(Q,x)
if (Q.head == Q.tail + 1) or (Q.head == 1 and Q.tail == Q.size) then
    error "overflow"
    return
Q[Q.tail] = x
if Q.tail == Q.size then
    Q.tail = 1
else  Q.tail = Q.tail + 1
```

صف

- در زیر تابع Dequeue پیاده سازی شده است.

Algorithm Dequeue

```
function DEQUEUE(Q)
if Q.head == Q.tail then
    error "underflow"
    return
x = Q[Q.head]
if Q.head == Q.size then
    Q.head = 1
else  Q.head = Q.head + 1
return x
```

صف

- یکی از کاربردهای مهم داده‌ساختار صف استفاده از آن در زمانبندی است.
- فرض کنید می‌خواهیم تعدادی واحد کاری [1] را در یک سیستم عامل زمانبندی کنیم. اولویت با واحدهای کاری است که زودتر وارد سیستم شده اند. می‌توانیم هر واحد کاری که وارد سیستم می‌شود را وارد صف کنیم و به ترتیب آنها را از صف خارج کرده، زمان پردازنده را به آنها اختصاص دهیم.

[1] task

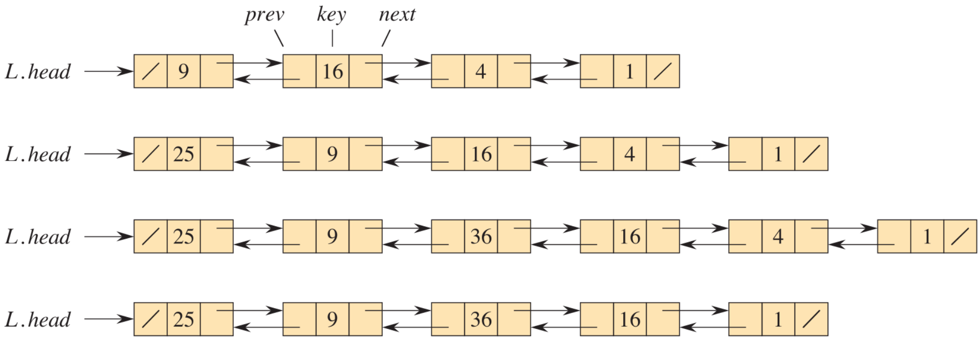
لیست پیوندی

- یک لیست پیوندی [1] داده ساختاری است که توسط آن مجموعه‌ای است که عناصر به صورت خطی مرتب شده‌اند به طوری که ترتیب عناصر در لیست با ترتیب مکان‌های حافظه عناصر الزاما یکسان نیست.
- برخلاف آرایه که در آن به عناصر با استفاده از اندیس آنها دسترسی پیدا می‌کنیم، در لیست پیوندی هر عنصر توسط یک اشاره‌گر به عنصر بعدی خود اشاره می‌کند و به هر عنصر می‌توان با استفاده از اشاره‌گری به آن دسترسی پیدا کرد.
- از آنجایی که در بسیاری مواقع عناصر لیست پیوندی دارای یک کلید و یک مقدار هستند، و می‌خواهیم به ازای یک کلید تعیین شده مقدار آن را پیدا کنیم، به لیست پیوندی، لیست جستجو [2] نیز گفته می‌شود.
- یک لیست پیوندی دو طرفه [3] یک لیست پیوندی است که عناصر آن علاوه بر ذخیره‌سازی عنصر بعدی، عنصر قبل خود را نیز ذخیره می‌کنند.

---
[1] linked list
[2] search list
[3] doubly linked list

لیست پیوندی

- در شکل زیر هریک از عناصر لیست پیوندی دو طرفه L یک ویژگی کلید (key) و دو اشاره‌گر برای تعیین عنصر قبل (prev) و عنصر بعد از خود (next) دارد. البته یک عنصر می‌تواند اطلاعات دیگری را نیز ذخیره کند.



لیست پیوندی

- به ازای عنصر داده شده x در لیست پیوندی، x.next به عنصر بعدی [1] و x.prev به عنصر قبلی [2] اشاره می‌کند.
- اگر x.prev=NIL باشد، آنگاه x عنصر ماقبل ندارد و در نتیجه اولین عنصر لیست یا عنصر ابتدای [3] لیست است.
- اگر x.next=NIL باشد، آنگاه x عنصر ما بعد ندارد و در نتیجه آخرین عنصر لیست یا عنصر انتهای [4] لیست است.
- ویژگی L.head به اولین عنصر لیست اشاره می‌کند. اگر L.head=NIL باشد، لیست تهی است.

---

[1] successor
[2] predecessor
[3] head
[4] tail

لیست پیوندی

- یک لیست پیوندی می‌تواند اشکال مختلفی داشته باشد. یک لیست می‌تواند یک طرفه [1] یا دو طرفه [2] باشد، می‌تواند مرتب شده یا غیر مرتب باشد، و همچنین می‌تواند حلقوی [3] یا غیر حلقوی باشد.
- اگر یک لیست پیوندی یک طرفه باشد، عناصر آن اشاره‌گر به عنصر بعدی دارند ولی اشاره‌گری به عنصر قبلی ندارند.

[1] singly
[2] doubly
[3] circular

لیست پیوندی

- اگر یک لیست مرتب شده باشد ترتیب خطی عناصر لیست متناسب با ترتیب خطی کلیدهای عناصر است بدین معنی که در لیست پیوندی مرتب شده صعودی همیشه مقدار کلید عنصر بعدی بزرگ‌تر یا مساوی مقدار کلید عنصر فعلی است و در لیست پیوندی مرتب شده نزولی همیشه مقدار کلید عنصر بعدی کوچک‌تر یا مساوی مقدار کلید عنصر فعلی است.
- در یک لیست پیوندی مرتب‌شده صعودی عنصر ابتدای لیست کمترین مقدار و عنصر انتهای لیست کمترین مقدار را دارد.
- اگر لیست پیوندی غیر مرتب [1] باشد، عناصر لیست با هر ترتیبی می‌توانند درکنار یکدیگر قرار گرفته باشند.

[1] unsorted

لیست پیوندی

- در یک لیست پیوندی حلقوی [1] ، اشاره‌گر prev از عنصر ابتدای لیست به عنصر انتهای لیست اشاره می‌کند و اشاره‌گر next از عنصر انتهای لیست به عنصر ابتدای لیست اشاره می‌کند.
- لیست‌هایی که در ادامه بررسی خواهیم کرد، غیر مرتب و دو طرفه هستند.

[1] circular linked list

لیست پیوندی

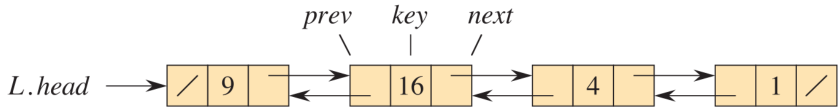
- جستجو در لیست پیوندی : تابع List-Search(L،k) اولین عنصر در لیست L با کلید k را توسط یک جستجوی خطی پیدا کرده، اشاره‌گری به عنصر یافته شده باز می‌گرداند. اگر هیچ عنصری با کلید k پیدا نشود، تابع مقدار NIL را باز می‌گرداند.

Algorithm List Search

```
   function List-Search(L,k)
1: x = L.head
2: while x ≠ NIL and x.key ≠ k do
3:    x = x.next
4: return x
```

لیست پیوندی

- در شکل زیر فراخوانی تابع `List-Search(L,4)` اشاره‌گری به سومین عنصر لیست باز می‌گرداند و فراخوانی تابع `List-Search(L,7)` مقدار NIL را باز می‌گرداند.



- برای جستجوی یک لیست با n عنصر، تابع List-Search در بدترین حالت در زمان $\Theta(n)$ اجرا می‌شود، زیرا نیاز دارد همه عناصر لیست را جستجو کند.

لیست پیوندی

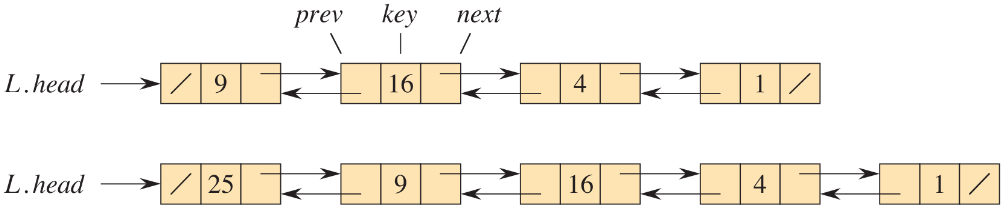
- درج در لیست پیوند : به ازای عنصر x که کلید آن تعیین شده است، تابع List-Prepend عنصر x را به ابتدای لیست پیوندی اضافه می‌کند.

Algorithm List Prepend

```
function List-Prepend(L,x)
x.next = L.head
x.prev = NIL
if L.head ≠ NIL then
   L.head.prev = x
L.head = x
```

لیست پیوندی

- در شکل زیر یک عنصر در لیست پیوندی درج شده است.



- توجه کنید که L.head.prev در واقع عنصر ماقبل عنصر ابتدای لیست است.
- زمان اجرای تابع List.Prepend بر روی یک لیست با n عنصر برابر با O(1) است.

لیست پیوندی

- درج در هر مکانی در لیست پیوندی می‌تواند انجام شود.
- اگر اشاره‌گری به عنصر y داشته باشیم، تابع List-Insert عنصر جدید x را به عنوان عنصر بعد از y در زمان $O(1)$ اضافه می‌کند.

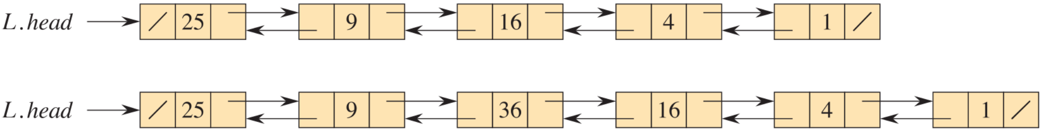
Algorithm List Insert

```
function List-Insert(x,y)
x.next = y.next
x.prev = y
if y.next ≠ NIL then
   y.next.prev = x
y.next = x
```

- از آنجایی که این تابع نیازی به دسترسی به لیست L ندارد، L به عنوان پارامتر به آن ارسال نشده است.

لیست پیوندی

- در شکل زیر عنصر ۳۶ بعد از عنصر ۹ اضافه شده است.



لیست پیوندی

- حذف از یک لیست پیوندی : تابع List-Delete عنصر x را از لیست پیوندی L حذف می‌کند.

Algorithm List Delete

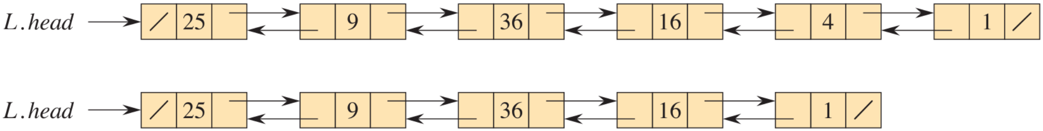
```
function LIST-DELETE(L,x)
if x.prev ≠ NIL then
   x.prev.next = x.next
else  L.head = x.next
if x.next ≠ NIL then
   x.next.prev = x.prev
```

- برای حذف یک عنصر با یک کلید معین، ابتدا تابع List-Search فراخوانی شده، اشاره‌گری به عنصر مورد نظر به دست می‌آید. سپس توسط تابع List-Delete عنصر مورد نظر از لیست حذف می‌شود.

- تابع List-Delete در زمان $O(1)$ اجرا می‌شود، اما برای حذف یک عنصر با یک کلید تعیین شده، ابتدا تابع List-Search در زمان $\Theta(n)$ باید اجرا شود.

لیست پیوندی

- در شکل زیر عنصر با کلید ۴ از لیست حذف شده است.



لیست پیوندی

- درج و حذف بر روی لیست پیوندی سریع‌تر از آرایه‌ها انجام می‌شوند.

- اگر بخواهیم یک عنصر به ابتدای یک آرایه اضافه کنیم یا عنصر اول را از آرایه حذف کنیم، آنگاه هریک از عناصر آرایه را باید یک خانه به سمت چپ یا راست منتقل کنیم.

- بنابراین در بدترین حالت درج و حذف در آرایه در زمان $\Theta(n)$ انجام می‌شود، درحالی‌که درج و حذف در لیست پیوندی در زمان $O(1)$ انجام می‌شود.

- از طرف دیگر دسترسی به عنصر k ام آرایه در زمان $O(1)$ انجام می‌شود، درحالی‌که زمان لازم برای دسترسی به عنصر k ام لیست پیوندی $\Theta(n)$ است.

- جستجو در هر دو داده‌ساختار آرایه و لیست پیوندی در بدترین حالت در زمان $\Theta(n)$ انجام می‌شود.

- پس به عناصر آرایه می‌توان سریع‌تر از لیست پیوندی دسترسی پیدا کرد، درحالی‌که حذف و درج در لیست پیوندی سریع‌تر از آرایه است.

لیست پیوندی

- تابع List-Delete را بسیار ساده‌تر می‌توان نوشت اگر شرایط مرزی را در ابتدا و انتهای لیست بررسی نکنیم.
- در این‌صورت تابع حذف را می‌توان به صورت زیر نوشت.

Algorithm List Delete′

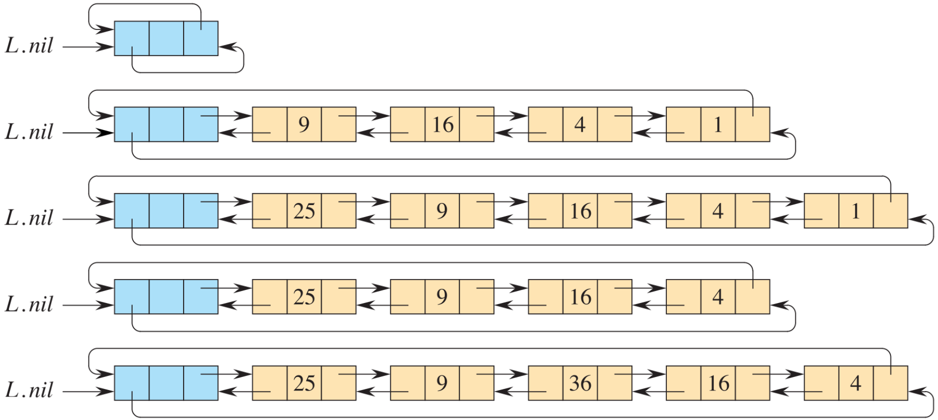
```
   function LIST-DELETE'(x)
1: x.prev.next = x.next
2: x.next.prev = x.prev
```

- نگهبان [1] به شیئی گفته می‌شود که بررسی شرایط مرزی را تسهیل می‌کند.

[1] sentinel

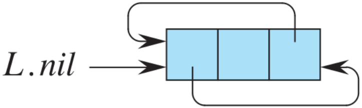
لیست پیوندی

- در شکل زیر برای تسهیل بررسی شرایط مرزی یک لیست پیوندی دو طرفه معمولی را به یک لیست پیوندی دو طرفه حلقوی با یک عنصر نگهبان تبدیل کرده‌ایم. عنصر L.nil یک شیء نگهبان است که نمایانگر تهی (NIL) است و همه ویژگی‌های عناصر دیگر لیست را داراست.



لیست پیوندی

- نگهبان L.nil در بین ابتدا و انتهای لیست قرار می‌گیرد. درواقع L.nil.next به ابتدای لیست اشاره می‌کند و L.nil.prev به انتهای لیست اشاره می‌کند. همچنین ویژگی next از عنصر انتهای لیست و ویژگی prev از عنصر ابتدای لیست هر دو به L.nil اشاره می‌کنند.
- از آنجایی که L.nil.next به عنصر ابتدای لیست اشاره می‌کند، ویژگی L.head را حذف می‌کنیم و با L.nil.next جایگزین می‌کنیم.
- یک لیست خالی به صورت زیر تنها حاوی عنصر نگهبان است.



لیست پیوندی

- با افزودن عنصر نگهبان، تابع حذف عنصر به صورت زیر نوشته می‌شود.

Algorithm List Delete′

```
function LIST-DELETE'(x)
x.prev.next = x.next
x.next.prev = x.prev
```

- در فرایند حذف عناصر هیچگاه عنصر نگهبان حذف نمی‌شود، مگر اینکه بخواهیم لیست را کاملا از بین ببریم.

لیست پیوندی

- تابع $\text{List–Insert}'$ عنصر x را در لیست بعد از y اضافه می‌کند.

Algorithm List Insert′

```
   function LIST-INSERT'(x,y)
1: x.next = y.next
2: x.prev = y
3: y.next.prev = x
4: y.next = x
```

لیست پیوندی

- برای جستجو در یک لیست پیوندی با نگهبان از L.nil.next آغاز می‌کنیم. اگر کلید مورد نظر در لیست وجود نداشته باشد، همهٔ عناصر لیست بررسی شده دوباره به L.nil باز می‌گردیم و در این صورت مقدار NIL را از تابع باز می‌گردانیم.

- تابع جستجو در لیست پیوندی با نگهبان به صورت زیر نوشته می‌شود.

Algorithm List Search′

```
function LIST-SEARCH'(L,k)
L.nil.key = k   ▷ store the key in the sentinel to guarantee it is in list
x = L.nil.next   ▷ start at the head of the list
while x.key ≠ k do
    x = x.next
if x == L.nil then   ▷ found k in the sentinel
    return NIL      ▷ k was not really in the list
else  return x
```

- نگهبان‌ها معمولاً کد را ساده می‌کنند و به مقدار ثابتی سرعت اجرای کد را کاهش می‌دهند اما مرتبه زمان اجرا را کاهش نمی‌دهند. دقت کنید در صورتی که بخواهیم از تعداد بسیار زیادی لیست‌های کوچک استفاده کنیم، نگهبان‌ها باعث می‌شوند فضای بسیار زیادی هدر رود. در این‌صورت بهتر است از نگهبان استفاده نکنیم.

حل روابط بازگشتی

- در مسئله جستجوی دودویی دیدیم چگونه می‌توان از روابط بازگشتی برای محاسبهٔ زمان اجرای الگوریتم‌ها بهره گرفت. در اینجا چند روش برای حل روابط بازگشتی مطرح می‌کنیم که عبارتند از روش جایگذاری [1] ، روش درخت بازگشت [2] و روش قضیه اصلی [3] .

[1] substitution method
[2] recursion-tree method
[3] master theorem method

روش جایگذاری

- روش جایگذاری برای حل روابط بازگشتی از دو گام تشکیل شده است. در گام اول جواب رابطهٔ بازگشتی یا عبارت فرم بسته [1] که در رابطهٔ بازگشتی صدق می‌کند حدس زده می‌شود. در گام دوم توسط استقرای ریاضی [2] اثبات می‌شود که جوابی که حدس زده شده است درست است و در رابطهٔ بازگشتی صدق می‌کند.
- برای اثبات توسط استقرای ریاضی، ابتدا باید ثابت کرد که جواب حدس زده شده برای مقادیر کوچک n درست است. سپس باید اثبات کرد که اگر جواب حدس زده شده برای n درست باشد، برای n+۱ نیز درست است. در این روش از جایگذاری جواب حدس زده شده در رابطهٔ اصلی برای اثبات استفاده می‌شود و به همین دلیل روش جایگذاری نامیده می‌شود.
- متاسفانه هیچ قاعدهٔ کلی برای حدس زدن جواب رابطهٔ بازگشتی وجود ندارد و یک حدس خوب به کمی تجربه و خلاقیت نیاز دارد.

[1] closed-form expression
[2] mathematical induction

روش جایگذاری

- برای مثال فرض کنید می‌خواهیم رابطهٔ $T(n) = 2T(n-1)$ و $T(0) = 1$ را حل کنیم.
- این رابطه را برای n های کوچک می‌نویسیم و حدس می‌زنیم $T(n) = 2^n$ باشد.
- سپس رابطه را با استفاده از استقرا اثبات می‌کنیم.

روش جایگذاری

- در برخی مواقع یک رابطۀ بازگشتی شبیه رابطه‌هایی است که جواب آنها را می‌دانیم و در چنین مواقعی می‌توانیم جواب را حدس بزنیم.

- برای مثال رابطه $T(n) = 2T(n/2 + 17) + \Theta(n)$ را در نظر بگیرید. فرض کنید می‌دانیم جواب رابطه $T(n) = 2T(n/2) + \Theta(n)$ برابر است با $T(n) = O(n \lg n)$. می‌توانیم حدس بزنیم که عدد ۱۷ برای n های بزرگ تأثیر زیادی ندارد. پس حدس میزنیم که جواب این رابطه $T(n) = O(n \lg n)$ باشد. سپس درستی این جواب را با استفاده از استقرا اثبات می‌کنیم.

- یک روش دیگر برای حدس زدن جواب این است که ابتدا یک کران پایین را حدس زده و سپس کران پایین را افزایش می‌دهیم تا به جواب واقعی نزدیک شویم.

روش درخت بازگشت

- روش دیگر برای حل مسائل بازگشتی، استفاده از درخت بازگشت [1] است.
- در این روش هر رأس از درخت، هزینه محاسبات یکی از زیر مسئله‌ها را نشان می‌دهد.
- هزینهٔ کل اجرای یک برنامه عبارت است از هزینه‌ای که در سطح صفر درخت نیاز است به علاوه هزینه محاسبه زیر مسئله‌ها. به همین ترتیب هزینه محاسبه هر یک از زیر مسئله‌های سطح اول تشکیل می‌شود از هزینه مسئله در سطح یک به علاوهٔ هزینهٔ زیر مسئله‌های سطح دوم و به همین ترتیب الی آخر.
- بنابراین اگر هزینهٔ محاسبهٔ همه رئوس درخت بازگشت را جمع کنیم، هزینه کل اجرای برنامه به دست می‌آید.

[1] recursion tree

روش درخت بازگشت

- رابطه بازگشتی زیر را در نظر بگیرید.

$$T(n) = 2T(n/2) + \Theta(n)$$
$$T(1) = \Theta(1)$$

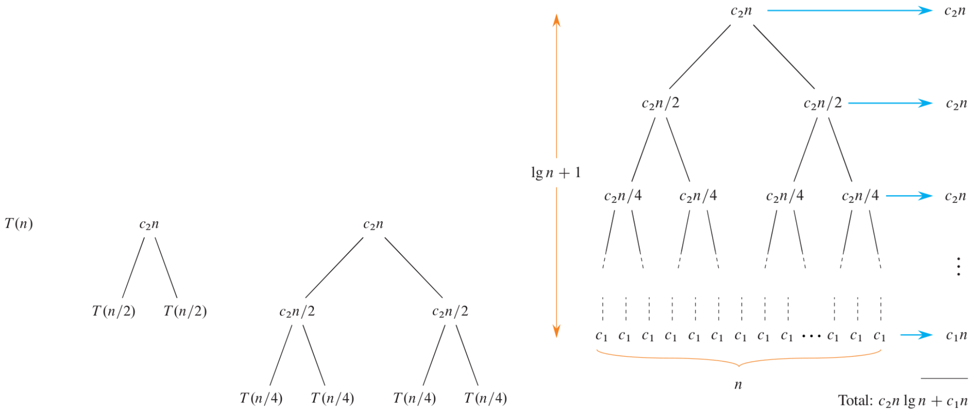
- برای سادگی فرض می‌کنیم طول آرایهٔ ورودی برابر با n بوده و n توانی از ۲ است. با این ساده‌سازی همیشه با تقسیم n بر ۲ یک عدد صحیح به دست می‌آید.

- زمان اجرای الگوریتم را به صورت زیر می‌نویسیم.

$$T(n) = \begin{cases} c_1 & \text{اگر } n = 1 \\ 2T(n/2) + c_2 n & \text{اگر } n > 1 \end{cases}$$

روش درخت بازگشت

- شکل‌های زیر محاسبه زمان اجرا را با استفاده از درخت بازگشت نشان می‌دهد.



روش درخت بازگشت

- زمان اجرا در هر یک از سطوح درخت برابر است با $c_2 n$.
- سطح آخر، یعنی سطحی که برگ‌های درخت در آن قرار دارد، حالت پایه را نشان می‌دهد که در این حالت زمان اجرا برابر است با c_1 و چون تعداد n زیر مسئله در این سطح ۱ داریم، زمان اجرای کل برابر است با $c_1 n$.
- از آنجایی که این درخت در هر مرحله به دو بخش تقسیم می‌شود، تعداد سطوح درخت برابر است با $\lg n + 1$.
- بنابراین زمان کل اجرای الگوریتم برابر است با $c_2 n \lg n + c_1 n$.
- می‌توانیم با استفاده از تحلیل مجانبی بنویسیم $T(n) = \Theta(n \lg n)$.

روش قضیه اصلی

- روش قضیه اصلی [1] برای حل مسائل بازگشتی استفاده می‌شود که به صورت $T(n) = aT(n/b) + f(n)$ هستند به طوری که $a > 0$ و $b > 1$ دو ثابت هستند.
- تابع $f(n)$ در اینجا تابع محرک [2] نامیده می‌شود و یک رابطهٔ بازگشتی که به شکل مذکور است، رابطهٔ بازگشتی اصلی [3] نامیده می‌شود.
- اگر یک رابطهٔ بازگشتی شبیه رابطه قضیه اصلی باشد و علاوه بر آن چند عملگر کف و سقف در آن وجود داشته باشد، همچنان می‌توان از رابطهٔ قضیه اصلی استفاده کرد.

[1] master theorem method
[2] driving function
[3] master recurrence

روش قضیه اصلی

- قضیه اصلی : فرض کنید $a > 0$ و $b > 1$ دو ثابت باشند و $f(n)$ یک تابع باشد که برای اعداد بسیار بزرگ تعریف شده باشد.
- رابطهٔ بازگشتی $T(n)$ که بر روی اعداد طبیعی $n \in \mathbb{N}$ تعریف شده است را به صورت زیر در نظر بگیرید.

$$T(n) = aT(n/b) + f(n)$$

روش قضیه اصلی

- رفتار مجانبی $T(n) = aT(n/b) + f(n)$ به صورت زیر است :

۱- اگر ثابت $\epsilon > 0$ وجود داشته باشد به طوری‌که $f(n) = O(n^{\log_b^a - \epsilon})$ آنگاه $T(n) = \Theta(n^{\log_b^a})$.

۲- اگر ثابت $k \geqslant 0$ وجود داشته باشد به طوری‌که $f(n) = \Theta(n^{\log_b^a} \lg^k n)$ آنگاه $T(n) = \Theta(n^{\log_b^a} \lg^{k+1} n)$.

۳- اگر ثابت $\epsilon > 0$ وجود داشته باشد به طوری‌که $f(n) = \Omega(n^{\log_b^a + \epsilon})$ آنگاه $T(n) = \Theta(f(n))$.
برای برخی از توابع $f(n)$ نیاز داریم بررسی کنیم $f(n)$ در رابطهٔ $af(n/b) \leqslant cf(n)$ به ازای $c < 1$ و n های به اندازهٔ کافی بزرگ صدق کند، اما برای توابعی که در تحلیل الگوریتم‌ها به آنها برمی‌خوریم این شرط معمولا برقرار است.

روش قضیه اصلی

- در یک حالت خاص اگر داشته باشیم، $T(n) = aT(n/b) + cn^k$ آنگاه می‌توانیم اثبات کنیم:

$$T(n) = \begin{cases} \Theta(n^{\log_b a}) & a > b^k \text{ اگر} \\ \Theta(n^k \lg n) & a = b^k \text{ اگر} \\ \Theta(n^k) & a < b^k \text{ اگر} \end{cases}$$

روش قضیه اصلی

- رابطهٔ بازگشتی $T(n) = 9T(n/3) + n$ را در نظر بگیرید. در این رابطه داریم $a = 9$ و $b = 3$ بنابراین به دست می‌آوریم $n^{\log_b^a} = n^{\log_3^9} = \Theta(n^2)$. از آنجایی که $f(n) = n = O(n^{2-\epsilon})$ به ازای هر ثابت $\epsilon \leqslant 1$ بنابراین می‌توانیم حالت اول در قضیه اصلی را در نظر بگیریم و نتیجه بگیریم $T(n) = \Theta(n^2)$.

روش قضیه اصلی

- رابطهٔ بازگشتی $T(n) = T(2n/3) + 1$ را در نظر بگیرید. در این رابطه داریم $a = 1$ و $b = 3/2$ بنابراین $n^{\log_b^a} = n^{\log_{3/2}^1} = n^0 = 1$. در اینجا حالت دوم در قضیه اصلی را داریم یعنی $f(n) = 1 = \Theta(n^{\log_b^a} \lg^0 n) = \Theta(1)$ بنابراین جواب رابطهٔ بازگشتی برابر است با $T(n) = \Theta(\lg n)$.

روش قضیه اصلی

- در رابطهٔ بازگشتی $T(n) = 3T(n/4) + n\lg n$ داریم $a = 3$ و $b = 4$ که بدین معنی است که $n^{\log_b^a} = n^{\log_4^3} = \Theta(n^{0.793})$. از آنجایی که $f(n) = n\lg n = \Omega(n^{\log_4^3 + \epsilon})$ جایی که ϵ حدود 0.2 است، بنابراین حالت سوم در قضیه اصلی را می‌توانیم در نظر بگیریم اگر شرط $af(n/b) \leqslant cf(n)$ برقرار باشد.

$$af(n/b) = 3(n/4)\lg(n/4) \leqslant (3/4)n\lg n = 3/4f(n)$$

بنابراین با استفاده از حالت سوم جواب رابطهٔ بازگشتی برابراست با $T(n) = \Theta(n\lg n)$.

روش قضیه اصلی

- رابطهٔ بازگشتی $T(n) = 2T(n/2) + \Theta(n)$ را در نظر بگیرید. از آنجایی که $a = 2$ و $b = 2$ داریم $n^{\log_2^2} = n$. حالت دوم در اینجا برقراراست زیرا به ازای $k = 0$ داریم $f(n) = \Theta(n)$ و بنابراین جواب رابطهٔ بازگشتی برابر است با $T(n) = \Theta(n \lg n)$.

روش قضیه اصلی

- رابطهٔ $T(n) = 8T(n/2) + \Theta(1)$ را در نظر بگیرید. در اینجا داریم $a = 8$ و $b = 2$ بنابراین $n^{\log_2^8} = n^3$. تابع محرک $f(n) = \Theta(1)$ است و بنابراین به ازای هر $\epsilon < 3$ داریم $f(n) = O(n^{3-\epsilon})$. بنابراین حالت اول قضیه اصلی برقرار است. نتیجه می‌گیریم $T(n) = \Theta(n^3)$.

روش قضیه اصلی

- در رابطهٔ بازگشتی $T(n) = 7T(n/2) + \Theta(n^2)$ داریم $a = 7$ و $b = 2$ بنابراین $n^{\log_2^7} = n^{\lg 7}$. از آنجایی که $\lg 7 = 2.8073...$ ، می‌توانیم قرار دهیم $\epsilon = 0.8$ و برای تابع محرک خواهیم داشت $f(n) = \Theta(n^2) = O\left(n^{\lg 7-\epsilon}\right)$ ، پس حالت اول در قضیه اصلی برقرار است و بنابراین جواب رابطهٔ بازگشتی برابر است با $T(n) = \Theta(n^{\lg 7})$.