

به نام خدا

ساختمان داده

آرش شفيعی



داده ساختارهای پایه

- در این بخش با چند داده ساختار پایه از جمله آرایه‌ها¹، ماتریس‌ها²، پشته‌ها³، صف‌ها⁴، و لیست‌های پیوندی⁵ آشنا خواهیم شد.

¹ arrays

² matrices

³ stacks

⁴ queues

⁵ linked lists

- 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 همه فضاها حافظه مستقیماً در یک زمان معین دسترسی پیدا کند، دسترسی به عناصر آرایه در زمان ثابت صورت می‌گیرد.

- 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.

- برای جستجوی یک مقدار در یک آرایه باید همهٔ عناصر آرایه را یک به یک 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 دودویی¹ استفاده کنیم.

¹ binary search

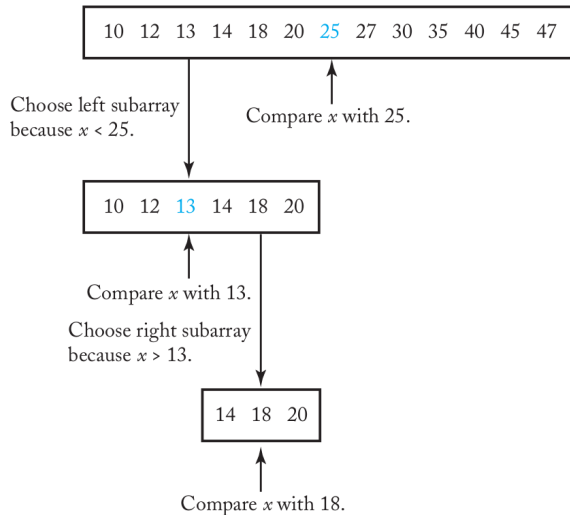
- الگوریتم جستجوی دودویی آرایه را به دو قسمت تقسیم می‌کند. برای جستجوی مقدار x در آرایه A ، ابتدا مقدار x با عنصر وسط آرایه یعنی $A[n/2]$ مقایسه می‌شود. اگر x برابر با مقدار وسط آرایه بود، مقدار مورد نظر یافته شده است. اگر x کوچکتر از عنصر وسط آرایه بود، باید x را در نیمه اول آرایه یعنی $A[1:n/2-1]$ 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.

- بنابراین مراحل انجام جستجوی دودویی به صورت زیر است.

۱. برای پیدا کردن مقدار x در آرایه $A[low:high]$ قرار می‌دهیم $mid = \lfloor (low + high) / 2 \rfloor$. 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 low از $high$ 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.

- برای پیدا کردن عدد ۱۸ در آرایه زیر، الگوریتم به صورت زیر عمل می کند.



- الگوریتم جستجوی دودویی به صورت زیر است.

Algorithm Binary Search

```
function BINARYSEARCH(A, x, low, high)
1: if (low > high) then
2:   return -1
3: mid =  $\lfloor (low + high) / 2 \rfloor$ 
4: if (x == A[mid]) then
5:   return mid
6: if (x < A[mid]) then
7:   return BinarySearch (A, x, low, mid-1)
8: else
9:   return BinarySearch (A, x, mid+1, high)
```

- برای جستجوی مقدار x جستجوی دودویی باید به صورت $\text{BinarySearch}(A, x, 1, n)$ فراخوانی شود.

- در تقسیم یک آرایه به دو قسمت صرفاً 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 $T(n) = O(\lg n)$.

- ماتریس یک آرایه دو بعدی است که می‌توانیم آن را توسط چند آرایه یک بعدی نمایش دهیم.
- دو روش معمول ذخیره ماتریس‌ها ترتیب سطری¹ و ترتیب ستونی² نام دارند.
- فرض کنید یک ماتریس با ابعاد $m \times n$ 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

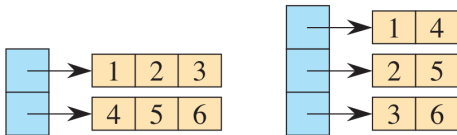
- برای مثال ماتریس $M = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$ با ابعاد 2×3 را در نظر بگیرید.
- در ترتیب سطری ماتریس به صورت 1, 2, 3 ; 4, 5, 6 در حافظه ذخیره می‌شود و در ترتیب ستونی ماتریس به صورت 1, 4 ; 2, 5 ; 3, 6 در حافظه ذخیره می‌شود.
- در شکل زیر نشان داده شده است که این ماتریس چگونه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

1	2	3	4	5	6
---	---	---	---	---	---

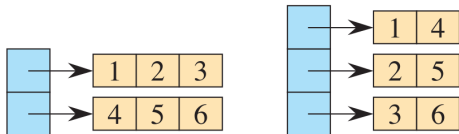
1	4	2	5	3	6
---	---	---	---	---	---

- 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 $4 = 3(2 - 1) + 1$ ذخیره می‌شود و با ترتیب ستونی در مکان $2 = 2 + 2(1 - 1)$ ذخیره می‌شود.

- 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.



- 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 سطرهاى 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.



- 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.

- اگر درایه‌های یک ماتریس اکثراً برابر با صفر باشند، به آن ماتریس یک ماتریس خلوت¹ گفته می‌شود.
- در مقابل، اگر درایه‌های یک ماتریس اکثراً غیر صفر باشند، به آن ماتریس یک ماتریس چگال¹ 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 غیر صفر را با ذکر شماره ستون آنها درج کنیم.

¹ sparse matrix

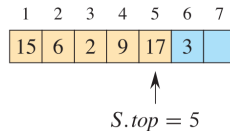
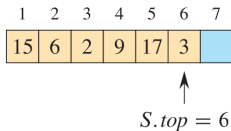
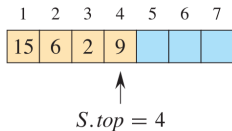
¹ dense matrix

- پشته¹ 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 LIFO² 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-in first-out

- شکل زیر چگونگی پیاده‌سازی پشته‌ای از n عنصر را در آرایه $S.data[1 : n]$ 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 اندازه آرایه‌ای است که پشته با استفاده از آن پیاده‌سازی شده است. عناصر پشته در S.data[1 : S.top] قرار می‌گیرند. عنصر S.data[1] عنصر انتهای¹ پشته و عنصر S.data[S.top] عنصر روی² پشته نامیده می‌شوند.

Data Structure Stack

```
struct STACK
1: int size
2: int top
3: T[] data  ► dynamically allocated array of type T
```

¹ bottom

² top

– ویژگی‌های پشته باید مقداردهی اولیه شوند.

Algorithm Stack Initialization

```
function STACK-INIT(S, n)
1: S.size = n
2: S.top = 0
3: S.data = new T[n]
```

- وقتی $S.top = 0$ است، پشته هیچ عنصری را شامل نمی‌شود و خالی¹ 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 خالی² مواجه می‌شویم. همچنین اگر $S.top$ بیشتر از $S.size$ شود با خطای سرریز پشته³ مواجه می‌شویم.

¹ empty

² underflow

³ overflow

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

Algorithm Stack Empty

```
function STACK-EMPTY(S)
1: if S.top == 0 then
2:   return true
3: else
4:   return false
```

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

Algorithm Push

```
function PUSH(S, x)
1: if S.top == S.size then
2:   error "overflow"
3: else
4:   S.top = S.top + 1
5:   S.data[S.top] = x
```

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

Algorithm Pop

```
function POP(S)
1: if Stack-Empty(S) then
2:   error "underflow"
3: else
4:   S.top = S.top - 1
5:   return S.data[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 پسوندی¹ عبارتی است که در آن عملگر بعد از عملوندها قرار می‌گیرد.
- در نشانه‌گذاری معمول که نشانه‌گذاری میانوندی¹ نامیده می‌شود، یک عملگر بین دو عملگر قرار می‌گیرد.
- برای مثال عبارت میانوندی $2 + 3$ در نشانه‌گذاری پسوندی به صورت $2\ 3\ +$ نوشته می‌شود.

¹ postfix notation

¹ infix notation

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

- مقدار یک عبارات پسوندی را می‌توان به صورت زیر با استفاده از یک پشته محاسبه کرد.
- 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 کاملاً 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.

- یک عبارت میانوندی را می‌توان با استفاده از یک پشته به عبارت پسوندی تبدیل کرد. فرض کنید یک عبارت از اعداد و عملگرهای جمع، تفریق، ضرب و تقسیم تشکیل شده است. الگوریتم زیر یک عبارت میانوندی را به پسوندی تبدیل می‌کند.
- به ازای هر عملوندی که از ورودی خوانده می‌شود، مقدار آن در خروجی نوشته می‌شود. به ازای هر عملگری که از ورودی خوانده می‌شود، عملگر وارد پشته می‌شود، اما قبل از اضافه کردن عملگر به پشته، تا وقتی که عملگر روی پشته اولویت بالاتر یا برابر داشته باشد، عملگر روی پشته از پشته خارج می‌شود و به خروجی اضافه می‌شود. هرگاه با عملگر پرانتز باز مواجه شدیم، آن را وارد پشته می‌کنیم، و هرگاه با عملگر پرانتز بسته مواجه شدیم، عملگرها را از پشته خارج کرده، به ورودی اضافه می‌کنیم، تا وقتی که عملگر پرانتز باز رسیده، عملگر پرانتز باز را نیز از پشته خارج می‌کنیم.
- 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.

- یکی دیگر از کاربردهای پشته، استفاده از آن در پشته فراخوانی¹ توابع برای فراخوانی‌های تودرتو یا فراخوانی‌های بازگشتی است.
- هر بار یک تابع فراخوانی می‌شود، در پشته فراخوانی قرار می‌گیرد تا وقتی که اجرای آن به اتمام برسد و از روی پشته فراخوانی برداشته شود.

¹ call stack

- 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.
- داده ساختار صف دقیقاً 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

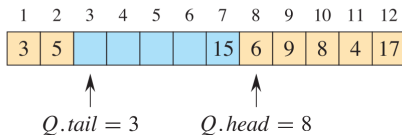
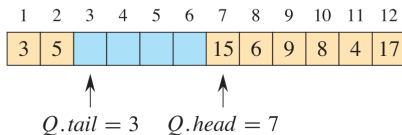
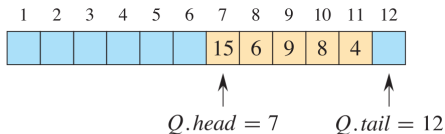
– یک صف شامل یک ابتدا¹ و یک انتها² است.

– 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

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



- ویژگی $Q.size$ اندازه صف است که برابر با طول آرایه 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 $Q.head + 1$ ، $Q.head + 2$ ، ... ، تا $Q.tail - 1$ قرار می‌گیرند.

Data Structure Queue

```

struct QUEUE
1: int size
2: int head
3: int tail
4: T[] data  ▷ dynamically allocated array of type T

```

- 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 خالی¹ 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 صف² مواجه می‌شویم.

¹ queue underflow

² queue overflow

- ویژگی‌های صف باید مقداردهی اولیه شوند.

Algorithm Queue Initialization

```
function QUEUE-INIT(Q, n)
1: Q.size = n
2: Q.head = 1
3: Q.tail = 1
4: Q.data = new T[n]
```

- در زیر تابع Enqueue پیاده سازی شده است.

Algorithm Enqueue

```
function ENQUEUE(Q,x)
1: if (Q.head == Q.tail + 1) or (Q.head == 1 and Q.tail == Q.size) then
2:   error "overflow"
3:   return
4: Q.data[Q.tail] = x
5: if Q.tail == Q.size then
6:   Q.tail = 1
7: else Q.tail = Q.tail + 1
```

- در زیر تابع Dequeue پیاده سازی شده است.

Algorithm Dequeue

```
function DEQUEUE(Q)
1: if Q.head == Q.tail then
2:   error "underflow"
3:   return
4: x = Q.data[Q.head]
5: if Q.head == Q.size then
6:   Q.head = 1
7: else Q.head = Q.head + 1
8: return x
```

- تعداد n رشته دودویی متوالی را با شروع از ۱ ، بدون عملیات جمع، با استفاده از یک صف چاپ کنید.
- عدد ۱ را وارد صف می‌کنیم.
- در هر گام عددی را از صف خارج کرده و چاپ می‌کنیم. سپس یک بار رقم صفر را به آن الحاق و حاصل را در صف وارد می‌کنیم و بار دیگر رقم یک را به آن الحاق و در صف وارد می‌کنیم. این کار را ادامه می‌دهیم تا تعداد n عدد چاپ شود.

- یکی از کاربردهای مهم داده ساختار صف استفاده از آن در زمانبندی است.
- فرض کنید می‌خواهیم تعدادی واحد کاری¹ را در یک سیستم عامل زمانبندی کنیم. اولویت با واحدهای کاری است که زودتر وارد سیستم شده‌اند. 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.
- تمرین: با استفاده از یک پشته، یک صف را وارونه کنید.

¹ task

لیست پیوندی

- یک لیست پیوندی¹ داده ساختاری است که توسط آن مجموعه‌ای از عناصر به صورت خطی مرتب شده‌اند به طوری که ترتیب عناصر در لیست با ترتیب مکان‌های عناصر در حافظه الزاماً 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 جستجو² نیز گفته می‌شود.
- یک لیست پیوندی دو طرفه³ یک لیست پیوندی است که عناصر آن علاوه بر ذخیره‌سازی عنصر بعدی، عنصر قبل خود را نیز ذخیره می‌کنند.

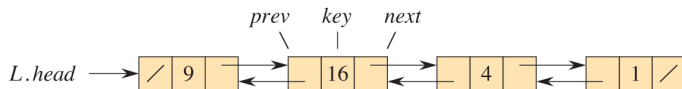
¹ linked list

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 اشاره گر برای تعیین عنصر قبل (prev) و عنصر بعد از خود (next) دارد. البته یک عنصر می تواند اطلاعات دیگری را نیز ذخیره کند.



- ساختمان داده‌های زیر، نحوه ذخیره سازی لیست پیوندی را نشان می‌دهد.

Data Structure Node

```
struct NODE
```

```
1: int key
```

```
2: T data    ▷ data of type T associated with a node
```

```
3: Node * next  ▷ pointer to the next node in the list
```

```
4: Node * prev  ▷ pointer to the previous node in the list
```

Data Structure Linked List

```
struct LINKEDLIST
```

```
1: Node * head  ▷ pointer to the head of the list
```

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 $x.prev = \text{NIL}$ باشد، آنگاه x عنصر ماقبل ندارد و در نتیجه اولین عنصر لیست یا عنصر ابتدای ³ لیست است.
- اگر $x.next = \text{NIL}$ 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 $L.head = \text{NIL}$ باشد، لیست تهی است.

¹ successor

² predecessor

³ head

⁴ tail

- یک لیست پیوندی می‌تواند اشکال مختلفی داشته باشد. یک لیست می‌تواند یک طرفه¹ یا دو طرفه² باشد، می‌تواند مرتب شده یا غیر مرتب باشد، و همچنین می‌تواند حلقوی³ 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

- 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 مرتب شده صعودی عنصر ابتدای لیست کمترین مقدار و عنصر انتهای لیست کمترین مقدار را دارد.
- اگر لیست پیوندی غیر مرتب¹ باشد، عناصر لیست با هر ترتیبی می‌توانند در کنار یکدیگر قرار گرفته باشند.

¹ unsorted

- در یک لیست پیوندی حلقوی¹ ، اشاره گر prev از عنصر ابتدای لیست به عنصر انتهای لیست اشاره می کند و اشاره گر next از عنصر انتهای لیست به عنصر ابتدای لیست اشاره می کند.
- لیست هایی که در ادامه بررسی خواهیم کرد، غیر مرتب و دو طرفه هستند.

¹ circular linked list

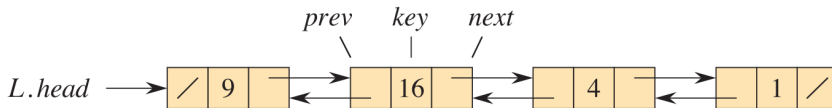
- جستجو در لیست پیوندی : تابع $\text{List-Search}(L, k)$ اولین عنصر در لیست L با کلید k را توسط یک جستجوی خطی پیدا کرده، اشاره‌گری به عنصر یافته شده باز می‌گرداند. اگر هیچ عنصری با کلید k پیدا نشود، تابع مقدار NIL را باز می‌گرداند.

Algorithm List Search

```
function LIST-SEARCH(L, k)
1: x = L.head
2: while x  $\neq$  NIL and x.key  $\neq$  k do
3:   x = x.next
4: return x
```

لیست پیوندی

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



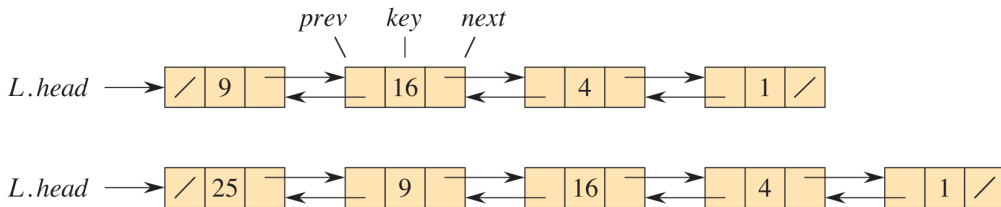
- برای جستجوی یک لیست با n عنصر، تابع $List\text{-}Search$ در بدترین حالت در زمان $\Theta(n)$ اجرا می‌شود، زیرا نیاز دارد همه عناصر لیست را جستجو کند.

- درج در لیست پیوندی : 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)
1: x.next = L.head
2: x.prev = NIL
3: if L.head  $\neq$  NIL then
4:   L.head.prev = x
5: L.head = x
```

- 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

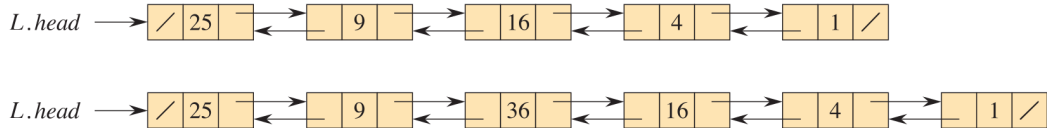
- درج در هر مکانی در لیست پیوندی می‌تواند انجام شود.
- اگر اشاره‌گری به عنصر y داشته باشیم، تابع List-Insert عنصر جدید x را به عنوان عنصر بعد از y در زمان $O(1)$ اضافه می‌کند.

Algorithm List Insert

```
function LIST-INSERT(x,y)
1: x.next = y.next
2: x.prev = y
3: if y.next  $\neq$  NIL then
4:   y.next.prev = x
5: y.next = x
```

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

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



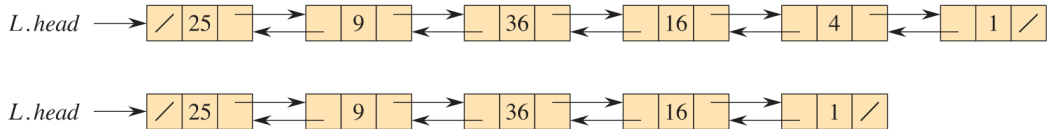
- حذف از یک لیست پیوندی : تابع List-Delete عنصر x را از لیست پیوندی L حذف می کند.

Algorithm List Delete

```
function LIST-DELETE(L,x)
1: if x.prev  $\neq$  NIL then
2:   x.prev.next = x.next
3: else L.head = x.next
4: if x.next  $\neq$  NIL then
5:   x.next.prev = x.prev
```

- برای حذف یک عنصر با یک کلید معین، ابتدا تابع List-Search فراخوانی شده، اشاره گری به عنصر مورد نظر به دست می آید. سپس توسط تابع List-Delete عنصر مورد نظر از لیست حذف می شود.
- تابع List-Delete در زمان $O(1)$ اجرا می شود، اما برای حذف یک عنصر با یک کلید تعیین شده، ابتدا تابع List-Search در زمان $\Theta(n)$ باید اجرا شود.

- 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، درحالی که درج و حذف در لیست پیوندی در زمان $\Theta(1)$ انجام می‌شود.
- از طرف دیگر دسترسی به عنصر k ام آرایه در زمان $\Theta(1)$ انجام می‌شود، درحالی که 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، درحالی که حذف و درج در لیست پیوندی سریع‌تر از آرایه است.

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

Algorithm List Delete'

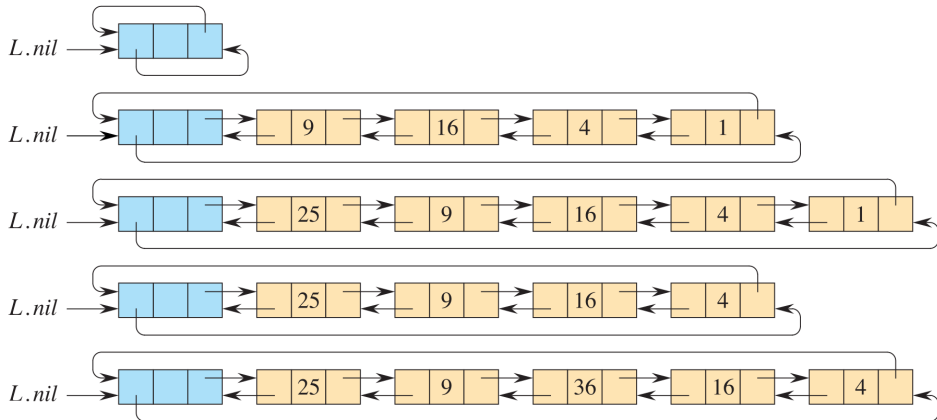
```
function LIST-DELETE'(x)
1: x.prev.next = x.next
2: x.next.prev = x.prev
```

- نگهبان¹ به شیئی گفته می‌شود که بررسی شرایط مرزی را تسهیل می‌کند.

¹ sentinel

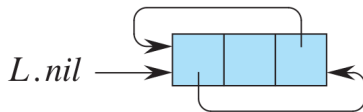
لیست پیوندی

- 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.



- ساختمان داده زیر، نحوه ذخیره سازی لیست پیوندی با عنصر نگهبان را نشان می دهد.

Data Structure Linked List'

```
struct LINKEDLIST'  
1: Node * nil  ▷ pointer to the sentinel of the list
```

- مقداردهی اولیه لیست پیوندی با عنصر نگهبان به صورت زیر است.

Algorithm Linked List Initialization'

```
function LINKEDLIST-INIT'(L)
1: L.nil = new Node()
2: L.nil.next = L.nil
3: L.nil.prev = L.nil
```

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

Algorithm List Delete'

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

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

- تابع 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)
1: L.nil.key = k  ▷ store the key in the sentinel to guarantee it is in
   list
2: x = L.nil.next  ▷ start at the head of the list
3: while x.key ≠ k do
4:   x = x.next
5: if x == L.nil then  ▷ found k in the sentinel
6:   return NIL      ▷ k was not really in the list
7: 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 این صورت بهتر است از نگهبان استفاده نکنیم.

- پیاده‌سازی پشته و صف با استفاده از لیست پیوندی نسبت به پیاده‌سازی توسط آرایه بهینه‌تر است، چراکه لیست پیوندی محدودیت اندازه ندارد. در پیاده‌سازی صف نیاز داریم دو اشاره‌گر به ابتدا و انتهای لیست پیوندی نگهداری کنیم. همچنین می‌توانیم از یک لیست پیوندی با عنصر نگهدارنده استفاده کنیم.
- لیست پیوندی نسبت به آرایه در درج و حذف سریع‌تر است، اما در دسترسی به عناصر کندتر است. لیست پیوندی همچنین محدودیتی بر روی تعداد عناصر اعمال نمی‌کند، با این حال سربار بیشتری نسبت به آرایه دارد چراکه هر بار تخصیص حافظه در هیپ برای هر عنصر جدید به زمان اندکی نیاز دارد. همچنین در استفاده از حافظه پنهان (کش)¹ آرایه بهینه‌تر است، چراکه حافظه آرایه پیوسته است و کل آرایه می‌تواند بر روی حافظه پنهان بارگیری شود.
- سیستم عامل برای نگهداری فضاهای خالی در حافظه هیپ، برای تخصیص پویای حافظه، از لیست پیوندی استفاده می‌کند.

¹ cache

- تمرین ۱: مرتب‌سازی درجی را توسط لیست پیوندی پیاده‌سازی کنید.
- تمرین ۲: پشته و صف را توسط لیست پیوندی پیاده‌سازی کنید.

- 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 جایگذاری¹، روش درخت بازگشت² و روش قضیه اصلی³.

¹ substitution method

² recursion-tree method

³ master theorem method

- روش جایگذاری برای حل روابط بازگشتی از دو گام تشکیل شده است. در گام اول جواب رابطه بازگشتی یا عبارت فرم بسته¹ که در رابطه بازگشتی صدق می کند حدس زده می شود. در گام دوم توسط استقرای ریاضی² اثبات می شود که جوابی که حدس زده شده است درست است و در رابطه بازگشتی صدق می کند.
- 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 $n+1$ نیز درست است. در این روش از جایگذاری جواب حدس زده شده در رابطه اصلی برای اثبات استفاده می شود و به همین دلیل روش جایگذاری نامیده می شود.
- متأسفانه هیچ قاعده کلی برای حدس زدن جواب رابطه بازگشتی وجود ندارد و یک حدس خوب به کمی تجربه و خلاقیت نیاز دارد.

¹ closed-form expression

² mathematical induction

- برای مثال فرض کنید می‌خواهیم رابطه $T(n) = 2T(n - 1)$ 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.

- در برخی مواقع یک رابطه 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.

- روش دیگر برای حل مسائل بازگشتی، استفاده از درخت بازگشت¹ است.
- در این روش هر رأس از درخت، هزینه محاسبات یکی از زیر مسئله‌ها را نشان می‌دهد.
- هزینه 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 علاوه هزینه زیر مسئله‌های سطح دوم و به همین ترتیب الی آخر.
- بنابراین اگر هزینه محاسبه 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.

$$\begin{aligned}T(n) &= 2T(n/2) + \Theta(n) \\T(1) &= \Theta(1)\end{aligned}$$

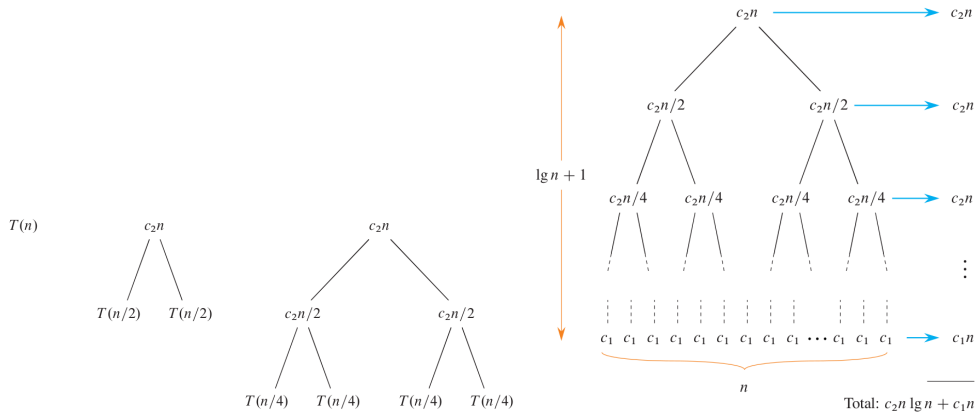
- برای سادگی فرض می‌کنیم طول آرایه ورودی برابر با n بوده و n توانی از ۲ است. با این ساده‌سازی همیشه با تقسیم n بر ۲ یک عدد صحیح به دست می‌آید.

- زمان اجرای الگوریتم را به صورت زیر می‌نویسیم.

$$T(n) = \begin{cases} c_1 & \text{اگر } n = 1 \\ 2T(n/2) + c_2n & \text{اگر } n > 1 \end{cases}$$

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_2n .
- 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_1n .
- 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 $c_2n \lg n + c_1n$.
- می‌توانیم با استفاده از تحلیل مجانبی بنویسیم $T(n) = \Theta(n \lg n)$.

- روش قضیه اصلی¹ 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 محرک² نامیده می‌شود و یک رابطه بازگشتی که به شکل مذکور است، رابطه بازگشتی اصلی³ نامیده می‌شود.
- در واقع رابطه بازگشتی اصلی زمان اجرای الگوریتم‌هایی را توصیف می‌کند که مسئله‌ای به اندازه n را به a زیر مسئله هر کدام با اندازه n/b تقسیم می‌کنند. تابع $f(n)$ هزینه تقسیم مسئله به زیر مسئله‌ها به علاوه هزینه ترکیب زیر مسئله‌ها را نشان می‌دهد.
- اگر یک رابطه بازگشتی شبیه رابطه قضیه اصلی باشد و علاوه بر آن چند عملگر کف و سقف در آن وجود داشته باشد، همچنان می‌توان از رابطه قضیه اصلی استفاده کرد.

¹ master theorem method

² driving function

³ master recurrence

- 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.
- رابطه بازگشتی $T(n)$ که بر روی اعداد طبیعی $n \in \mathbb{N}$ تعریف شده است را به صورت زیر در نظر بگیرید.
$$T(n) = aT(n/b) + f(n)$$

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

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

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

۳- اگر ثابت $\epsilon > 0$ وجود داشته باشد به طوری که $f(n) = \Omega(n^{\log_b^a + \epsilon})$ آنگاه $T(n) = \Theta(f(n))$.
 برای برخی از توابع $f(n)$ نیاز داریم بررسی کنیم $f(n)$ در رابطه $af(n/b) \leq cf(n)$ به ازای $c < 1$ و n های به اندازه کافی بزرگ صدق کند، اما برای توابعی که در تحلیل الگوریتم ها به آنها برمی خوریم این شرط معمولاً برقرار است.

- ۱- در حالت اول رشد جزء بازگشتی از رشد تابع محرک بیشتر است. به عنوان مثال در $T(n) = 2T(n/2) + \lg n$ رشد جزء بازگشتی $\Theta(n)$ و رشد تابع محرک $\Theta(\lg n)$ است. بنابراین $T(n) = \Theta(n)$.
- ۲- در حالت دوم رشد جزء بازگشتی و تابع محرک برابر است و یا رشد تابع محرک با یک ضریب $\Theta(\lg^k n)$ از جزء بازگشتی سریع تر است. به عنوان مثال در $T(n) = 2T(n/2) + n \lg n$ رشد جزء بازگشتی $\Theta(n)$ و رشد تابع محرک $\Theta(n \lg n)$ است. در این حالت $T(n) = \Theta(n \lg^2 n)$.
- ۳- در حالت سوم رشد جزء بازگشتی از رشد تابع محرک کمتر است. به عنوان مثال در $T(n) = 2T(n/2) + n^2$ رشد جزء بازگشتی $\Theta(n)$ و رشد تابع محرک $\Theta(n^2)$ است. بنابراین $T(n) = \Theta(n^2)$.

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

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

- رابطه 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 $\epsilon < 1$ بنابراین می‌توانیم حالت اول در قضیه اصلی را در نظر بگیریم و نتیجه بگیریم $T(n) = \Theta(n^2)$.

- رابطه 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 رابطه بازگشتی برابر است با $T(n) = \Theta(\lg n)$.

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

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

بنابراین با استفاده از حالت سوم جواب رابطه بازگشتی برابر است با $T(n) = \Theta(n \lg n)$.

- رابطه بازگشتی $T(n) = T(n/2) + \Theta(1)$ رابطه‌ای بود که برای جستجوی دودویی به دست آوردیم. از آنجایی که $a = 1$ و $b = 2$ داریم $n^{\log_2 1} = 1$. حالت دوم در اینجا برقرار است زیرا به ازای $k = 0$ داریم $f(n) = \Theta(1)$ و بنابراین جواب رابطه بازگشتی برابر است با $T(n) = \Theta(\lg n)$.