ساختمان داده ها

داده هایی که می خواهند پردازش شوند باید طوری سازماندهی شوند که روابط بین عناصر داده ها را نشان می دهند تا پردازش داده ها به طور کارآمدی انجام شود.

حل مساله شامل پردازش داده ها است و بخش مهمی از حل مسئله سازمان دهی دقیق داده ها است. که برای انجام این کار باید موارد زیر را مشخص کنیم:

۱- مجموعه ای از عناصر داده ها
۲- رابطه اصلی بین عناصر داده ها و عملیاتی که باید بر روی آنها انجام شوند.

این مجموعه از داده ها همراه با عملیات و رابطه ها، نوع داده انتزاعی (Abstract Data Type) نامیده می شود که به صورت خلاصه ADT نامیده می شود.

ساختمان داده ها

❖ منظور از کلمه انتزاعی این است که:
داده ها و عملیات اصلی و رابطه های تعریف شده بر روی آن مستقل از پیاده سازی آنها مورد مطالعه قرار می گیرد. راجع به آنچه باید روی داده ها انجام شود فکر می کنیم نه روی چگونگی انجام کار.
ایده انتزاع داده ها که در آن تعریف نوع داده مستقل از پیاده سازی آن انجام می شود مفهوم مهمی در نرم افزار است که ساختمان داده را بدون پرداختن به جزئیات پیاده سازی آن مطالعه و مورد استفاده قرار می دهد.

❖ پیاده سازی یک ADT شامل سه چیز است:
۱- ساختار حافظه که معمولا ساختمان داده نامیده می شود و برای ذخیره عناصر داده ها به کار می رود
۲- الگوریتمهایی که عملیات اصلی را انجام می دهند
۳- رابطه ها

کارایی الگوریتم

کارایی یک الگوریتم بر اساس دو معیار اندازه گیری می شود.
✓ بهره وری فضا (space utilization): میزان حافظه ای که برای ذخیره داده ها لازم است.
✓ کارایی زمان (time efficiency): زمان لازم برای پردازش داده ها است.

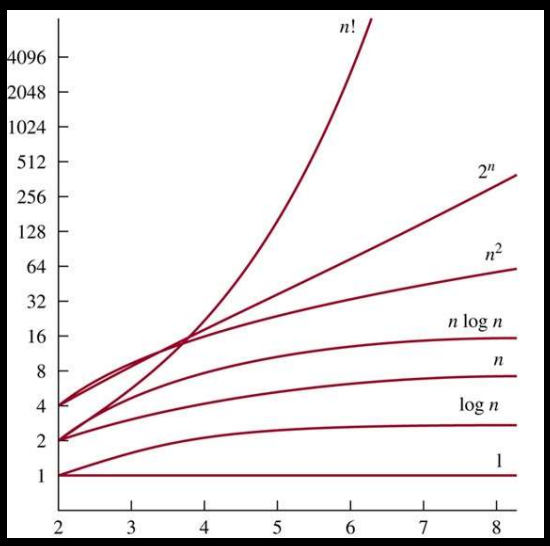
همیشه امکانپذیر نیست هم نیازمندیهای فضا و هم نیازمندیهای زمان را کم کرد. الگوریتم هایی که به فضای کمتری نیاز دارند، اغلب نسبت به آنهایی که به فضای بیشتری نیاز دارند کندتر هستند.
زمان اجرای الگوریتم تحت تاثیر عوامل متعددی است. یکی از عوامل، اندازه ورودی است زیرا تعداد مقادیر ورودی در زمان لازم برای پردازش آنها تاثیر دارد. به عنوان مثال زمان لازم برای مرتب سازی لیستی از مقادیر به تعداد عناصر لیست بستگی دارد.
زمان اجرای T برای یک الگوریتم باید به صورت یک تابع T(n) بیان شود که n اندازه ورودی است. T(n) تعداد تقریبی از دستوراتی است که باید اجرا شوند.

پیچیدگی

- مرتبه یک الگوریتم با بزرگترین جمله تابع پیچیدگی زمانی آن تعیین می شود.
- الگوریتم هایی با پیچیدگی زمانی از قبیل n، 100n را الگوریتم های خطی می گویند.
- برای توصیف زمان اجرای الگوریتم و مقایسه الگوریتم های مختلف از نمادهای $O, o, \Omega, \omega, \theta$ مجانبی استفاده می شود.

- در محاسبه O
  - ✓ فاکتورهای ثابت محاسبه نمی شوند.
  - ✓ نرخ رشد در چند جمله ای ها جمله با بیشترین توان است.
  - ✓ توان های بالاتر سریعتر از توان های پایین تر رشد می کنند.
  - ✓ توابع نمایی سریعتر از توابع توانی رشد می کنند.
  - ✓ توابع لگاریتمی کندتر از توابع توانی رشد می کنند.
  - ✓ تمامی توابع لگاریتمی رشدی مشابه دارند.
- محاسبه پیچیدگی حلقه for (i=x; i<=y ;i+=z) برابر است با $\left\lceil \frac{y-x+1}{z} \right\rceil$

ترتیب توابع رشد



- ترتیب توابع رشد به صورت زیر است:

$$O(1) < O(logn) < O(\sqrt{n}) < O(n) < O(nlogn) < O(n^2) < O(n^2logn) < O(n^3) < O(2^n) < O(3^n) < O(n!) < O(n^n)$$

پیچیدگی الگوریتم o(n)

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
        int x,i;                    0
        x=0;                        1
        for (i=1; i<=n ;i++)        n+1
                x+=i;               n
        cout<<x;                    1
        getch();                    1
        }                           2n+4   ⇒ o(n)
        system("pause");
        return 0;
}
```

پیچیدگی الگوریتم o(n)

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
        for (i=1; i<=56 ;i++)       o(1)
        {
                ******;
        }
        for (i=1; i<=n ;i++)        o(n)
        {
                ******;
        }
        system("pause");
        return 0;
}
```

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
        for (i=1; i<=n ;i++)        o(n²)
        {
                for (j=1; j<=m ;j++)
                        ******;
        }
        system("pause");
        return 0;
}
```

پیچیدگی الگوریتم o(n)

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
        j=n;
        while(j>1)
        {
                *****;
                j/=2;
        }
        system("pause");
        return 0;
}
```

o(logn)

```
#include "stdafx.h"
#include <iostream>

using namespace std;

int main()
{
        J=n;
        while(j>1)
        {
        for(i=1,i<=n,i++)
                *****;
        j/=2;
        }
        system("pause");
        return 0;
}
```

o(nlogn)

آرایه

علاوه بر انواع داده ساده، اغلب زبانهای برنامه سازی انواع داده ساختاری را تدارک می بینند. مقادیر این انواع به صورت مجموعه اند نه به صورت منفرد. یکی از متداولترین نوع داده ساختاری آرایه است.
تعداد عناصر آرایه ثابت است و عناصر دارای ترتیب هستند. عناصر آرایه باید همنوع باشند. می توان به عناصر آرایه دستیابی مستقیم داشت.

- دستیابی تصادفی یا مستقیم: هر عنصر آرایه می تواند با مشخص کردن محل آن در آرایه دستیابی شود.
- دستیابی ترتیبی: دستیابی به هر عنصر زمانی امکانپذیر است که عناصر قبلی دستیابی شوند.

اگر در آرایه از یک اندیس استفاده شود آرایه یک بعدی نامیده می شود و اگر بیش از یک اندیس داشته باشد، آرایه چندبعدی نامیده می شود.

آرایه

✓ برای تعریف آرایه از فرمول زیر استفاده می شود:

نوع_عناصر نام_آرایه [تعداد_عناصر_آرایه] ;

نوع_عناصر نام_آرایه [تعداد_عناصر_آرایه] ={لیست_مقادیر};

مثال:

```
int a[10];
int b={4,7,3};
```

مقداردهی اولیه

✓ آرایه ها بلوکی از محلهای متوالی حافظه را به خود اختصاص می دهند. اندیس آرایه ها از صفر شروع می شود.

✓ زمانیکه مقداردهی اولیه صورت میگیرد، اگر تعداد عناصر آرایه هم مشخص باشد، عناصری که مقدار ندارند، مقدارشان صفر است.

```
int b[5]={4,7,3};
```

4	7	3	0	0

✓ روش زیر راهی برای صفر کردن مقادیر تمام عناصر آرایه است.

```
int a[10]={0};
```

0	0	0	0	0	0	0	0	0	0

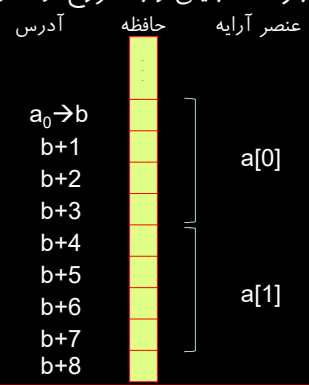
آرایه

آدرس اولین عنصر آرایه آدرس پایه (base address) نام دارد و نام آرایه به این عنصر اشاره دارد. آدرس سایر عناصر آرایه به صورت آفستی (offset) از این آدرس پایه مشخص می شود.

a[i] در یک بلوک n بایتی و با شروع از آدرس b+n*i ذخیره می‌شود. a[i]=b+n*i

n= sizeof(element_type)

b= base address



فرض می شود مقادیر صحیح در ۴ بایت ذخیره می شوند.

آرایه

- مشکلات آرایه:
  - ✓ کامپایلر حد آرایه ها را کنترل نمی کنند.
  - ✓ ظرفیت آرایه در طول اجرای برنامه قابل تغییر نیست.
  - ✓ آرایه ها self-contain نیستند. هر عملی که می خواهد روی آرایه ها انجام شود حداقل دو نوع اطلاعات نیاز دارد: خود آرایه و راههایی برای تعیین اندازه آرایه (تعداد مقادیری که در آن ذخیره شده اند) یا راهی برای یافتن آخرین مقدار موجود در آرایه.

یکی از قواعد برنامه نویسی شیء گرا این است که هر شیء باید self-contain باشد به این معنی که اطلاعاتی را که برای توصیف آن و انجام عملیات روی آن لازمند با خودش داشته باشد. آرایه ها اندازه و ظرفیت را همراه خودشان ندارند.

آرایه

تعداد عناصر آرایه های یک بعدی:

a[i] تعداد عناصر= i

تعداد عناصر آرایه های دو بعدی:

a[m][n] تعداد عناصر= m*n

تعداد عناصر آرایه های سه بعدی:

a[i][j][k] تعداد عناصر= i*j*k

یک روش تصور آرایه چندبعدی آرایه ای از آرایه هاست. پیاده سازی آرایه های چندبعدی پیچیده تر از آرایه های یک بعدی است. حافظه به صورت دنباله ای از محل های حافظه سازماندهی می شود، بنابراین ماهیت آن یک بعدی است.

آرایه

مثال:

```
char a[3][4];
```

هر کاراکتر در یک بایت ذخیره می شود، کامپایلر ۱۲ بایت متوالی را برای ذخیره عناصر آرایه رزرو می کنند.
عناصر آرایه ممکن است به صورت سطری (row major / rowwise) یا ستونی (column major / columnwise) ذخیره شوند.
در صورتی که به صورت سطری ذخیره شود ۴ بایت اول از آدرس پایه a برای ذخیره سطر اول، چهار بایت دوم برای ذخیره سطر دوم و چهار بایت سوم برای ذخیره سطر سوم مورد استفاده قرار می گیرد.
در صورتی که به صورت ستونی ذخیره شود ۳ بایت اول از آدرس پایه a برای ذخیره ستون اول، ۳ بایت دوم برای ذخیره ستون دوم و ۳ بایت سوم برای ذخیره ستون سوم و ۳ بایت چهارم برای ذخیره ستون آخر مورد استفاده قرار می گیرد.

آرایه

مثال: ماتریس زیر را در نظر بگیرید.

$$\begin{bmatrix} A & B & C & D \\ E & F & G & H \\ I & J & K & L \end{bmatrix}$$

سطری

A	B	C	D	E	F	G	H	I	J	K	L

ستونی

A	E	I	B	F	J	C	G	K	D	H	L

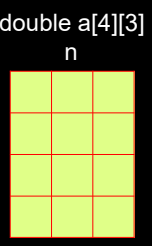
آرایه

برای اینکه مشخص شود عنصری از یک آرایه دو بعدی a[m][n] در کجا ذخیره شده است از روش زیر استفاده می شود.

آرایه a با m سطر و n ستون که هر عنصر آن به w بایت حافظه نیاز دارد و آدرس پایه b است را در نظر بگیرید:

آدرس (سطری) a[i][j]=i*n*w+j*w+b= b+(i*n+j)*w

آدرس (ستونی) a[i][j]=j*m*w+i*w+b= b+(j*m+i)*w

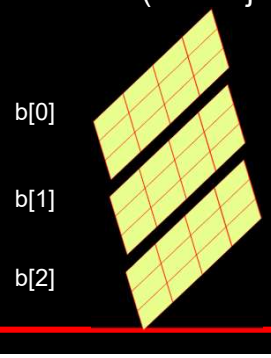


آرایه

برای اینکه مشخص شود عنصری از یک آرایه سه بعدی a[L][m][n] در کجا ذخیره شده است از روش زیر استفاده می شود.
آرایه a دارای L عنصر است که هر کدام آرایه دوبعدی هستند که m سطر و n ستون دارند. هر عنصر آن به w بایت حافظه نیاز دارد و آدرس پایه b است.

آدرس a[i][j][k]=i*m*n*w+j*n*w+k*w+b= b+(i*m*n+j*n+k)*w

double b[3][4][3];



ماتریس اسپارس

برای ذخیره یک ماتریس m*n می توان از یک آرایه دو بعدی با m سطر و n ستون استفاده کرد. گروهی از ماتریسها وجود دارند که به آن ماتریس خلوت یا اسپارس (sparse) گفته می شود. در این ماتریس ها اکثریت عناصر مقدار صفر دارند.

از آنجاییکه ماتریس های اسپارس در عمل وجود دارند و برخی موارد اندازه های آنها بسیار بزرگ است بایدروش بهینه تری برای ذخیره آنها در کامپیوتر ارائه شود. یک روش آن است که از یک آرایه دو بعدی با سه ستون استفاده کنیم. ستون های اول و دوم این آرایه موقعیت سطروستون در ماتریس اسپارس رانشان می دهند و ستون سوم مقدار ذخیره شده در آن سطر و ستون رانشان می دهد (تعداد سطرهای آین آرایه به تعداد مقدار ذخیره شده در ماتریس اصلی است)

ماتریس اسپارس

برای پیاده سازی ماتریس اسپارس می توان از یک ساختار کمک گرفت:

```
struct elements
 {
        int row, col ,value;
 };
void main()
{
    elements s[max];
    s[0].row=0;
    s[0].col=0;
    s[0].value=7;
}
```

	سطر	ستون	مقدار
	0	1	2
0	0	0	7
1	0	5	10
2	1	1	2
3	3	0	9
4	3	2	4
5	4	2	5
6	5	5	20

	0	1	2	3	4	5	6
0	7	0	0	0	0	10	0
1	0	2	0	0	0	0	0
2	0	0	0	0	0	0	0
3	9	0	4	0	0	0	0
4	0	0	5	0	0	0	0
5	0	0	0	0	0	20	0

ماتریس اسپارس

ماتریس اسپارس

- ترانهاده ماتریس اسپارس (ماتریس معکوس)

برای بدست آوردن ترانهاده باید جای سطرو ستون ها راعوض کرد، عنصری که در مکان [i][j] ازماتریس اولیه قرار دارد به مکان [j][i] درماتریس ثانویه منتقل میشود.
در ماتریس اسپارس ساده ترین راه جهت تولید معکوس آن این است که ابتدا عناصری که در ستون صفر ماتریس قرار دارند را به ترانهاده منتقل کرد به گونه ای که جای سطروستون عوض شود سپس برای ستون یک و غیره این کار انجام شود.

سطر	ستون	مقدار
0	0	7
1	5	10
1	1	2
3	0	9
3	2	4
4	2	5
5	2	20

ترانهاده

سطر	ستون	مقدار
0	0	7
0	3	9
1	1	2
2	3	4
2	4	5
3	0	10
5	5	20

ماتریس مثلثی

- ماتریس پایین مثلثی: ماتریس مربعی که تمام عناصر بالای قطر اصلی صفر باشند.
- ماتریس بالا مثلثی: ماتریس مربعی که تمام عناصر پایین قطر اصلی صفر باشند.
- در این ماتریس ها فقط عناصر غیرصفر ذخیره می شوند.
- در ماتریس مثلثی تعداد عناصر غیر صفر برابر است با $\frac{n(n+1)}{2}$
- در ماتریس مثلثی تعداد عناصر صفر برابر است با $\frac{n(n-1)}{2}$

پایین مثلثی

3	0	0	0
1	6	0	0
6	1	2	0
7	1	4	9

بالا مثلثی

3	1	9	7
0	6	8	5
0	0	2	4
0	0	0	9

ساختار

گاهی انواع داده موجود در زبان برای پیاده سازی عملیات برنامه کافی نیست و باید انواع داده جدیدی را تعریف کرد. برای تعریف انواع جدید می توان از ساختار، یونیون یا کلاس بر اساس نیاز مسئله استفاده کرد.
ساختار یک نوع داده جمعی است. هر ساختار از یک یا چند عضو که به همراه هم یک واحد منطقی را می سازد تشکیل می شود. برخلاف آرایه ها که در آنها همه عناصر از یک نوع واحد هستند، هر عضو یک ساختار می تواند نوع داده خاص خود را داشته باشد که با نوع داده بقیه اعضا فرق داشته باشد.

```
struct name{
    type member1;
    type member 2;
    .
    .
    .
}variables;
```

ساختار

```
# include <iostream.h>
  struct students{
  char name[10];
  char family[20];
  int age;
  int entry;
  } st1,st2;

int main()
{
 strcpy(st1.name,"Mike");
 strcpy(st2.name,"Nick");
 st1.age=18;
 st2.age=20;
 cout << st1.age<<st1.name;
 return 0;
}
```

مقدار فضای ذخیره آرایه ای به طول ۲۰ برای ساختار روبرو برابر است با: (درصورتی که int دو بایت باشد)

(10+20+2+2)*20=680

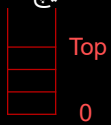
پشته Stack

مجموعه ای از عناصر مرتب که فقط از یک طرف قابل دستیابی هستند که بالای پشته نام دارد. عملیات اصلی شامل ساخت پشته، بررسی اینکه پشته خالی است، افرودن عنصری به بالای پشته (Push)، بازیابی عنصر بالای پشته (Top)، حذف عنصر بالای پشته (Pop)

پشته را ساختمان داده LIFO (Last In First Out) نیز می گویندو معنایش این است آخرین ورودی، اولین خروجی است.

پشته زمان اجرا برای نگهداری محیطهای توابع در فراخوانیهای آنها به کار می رود. درک چگونگی به کارگیری پشته درفراخوانیهای تابع،کلیددرک توابع بازگشتی است.

در زبان C++ نوع داده ای مثل پشته وجود ندارد و باید به وسیله ساختارهای موجود پشته را پیاده سازی کرد. به عنوان مثال می توان از یک آرایه جهت نگهداری عناصر پشته استفاده کرد. پشته ای که در آن هیچ عنصری وجود نداشته باشد پشته خالی نام دارد.



پشته Stack

اعمالی که روی پشته می توان انجام داد:

۱- افزودن عنصرجدید به بالای پشته Push: برای انجام این عمل به دوپارامترنیاز است یکی نام پشته ای که قرار است در آن درج شود و دوم نام عنصری که قرار است در پشته درج شود.

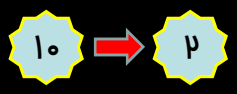
۲- عمل حذف ازبالای پشته Pop: برای انجام این عمل فقط نیاز به دانستن نام پشته است.

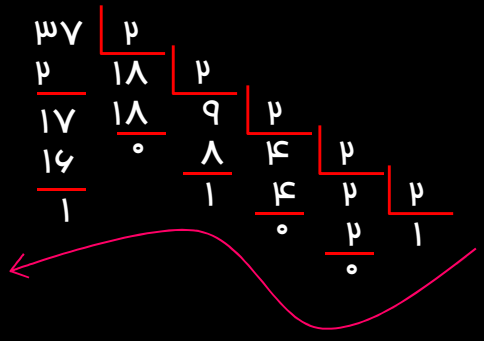
- ✓از دیدگاه تئوری پشته یک فضای نامحدود است که می توان به هر تعداد عنصر در آن درج کرد، اما در عمل به علت محدود بودن فضا چنین چیزی امکان پذیر نیست.
- ✓نمی توان از یک پشته خالی عمل حذف را انجام داد بنابراین برا ی حذف باید از خالی نبودن پشته اطمینان حاصل شود. هر تلاش برای حذف از پشته خالی منجر به وقوع رخدادی به نام under flow می شود و هر تلاش برای درج در پشته پر موجب Overflow می شود.

تبدیل یک عدد دسیمال به باینری با استفاده از Stack

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int Top, A[50];
int push(int x);
int pop(int &x);
int main()
{
        int Number, R;
        Top=-1;
        cin>>Number;
        while(Number!=0)
        {
                R=Number%2;
                push(R);
                Number=Number/2;
        }
        for(int i=Top; i>=0; i--)
        {
                pop(R);
                cout<<R;
        }
        system("pause");
        return 0;
}

void push(int x)
{
        if(Top>=49)
                return -1;
        Top++;
        A[Top]=x;
}

void pop(int &x)
{
        if(Top<0)
                return -1;
        x=A[Top];
        Top--;
}
```

تبدیل یک عدد دسیمال به باینری با استفاده از Stack

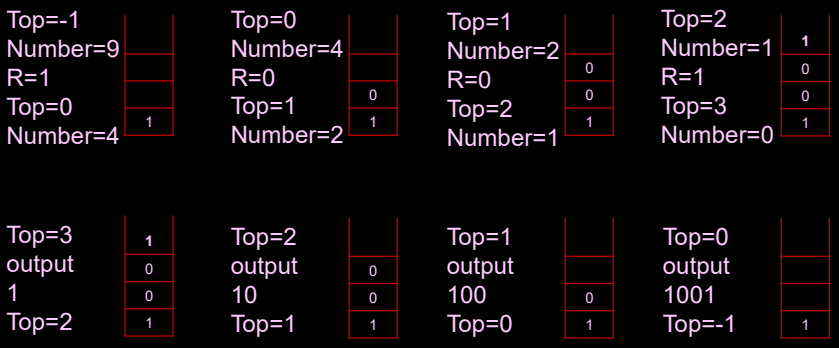




$(۳۷)_{۱۰} = (۱۰۰۱۰۱)_{۲}$

تبدیل یک عدد دسیمال به باینری با استفاده از Stack

مثال عددی



کاربرد پشته

در فراخوانیهای تابع از پشته استفاده می شود.
در ارزیابی عبارات نیز از پشته استفاده می شود. در اغلب زبانهای برنامه سازی عبارات محاسباتی به شکل میانوند (infix) نوشته می شوند. عبارات را می توان به صورت پسوند (postfix) و پیشوند (prefix) نیز نمایش داد.
اغلب کامپایلرها عبارات میانوند را به شکل عبارات پسوند در می آورند و سپس دستورات زبان ماشین را برای ارزیابی آنها تولید می نماید، زیرا تبدیل عبارات میانوند به پسوند و ارزیابی این عبارات ساده است. عبارات میانوند می توانند حاوی پرانتز باشند تا تقدم عملگرها را مشخص کنند ولی عبارات پسوندی فاقد پرانتزند.
برای تبدیل عبارات میانوند به پسوند و ارزیابی آنها نیاز به استفاده از پشته است.

A+B	infix
AB+	postfix
+AB	prefix

کاربرد پشته

برای بدست آوردن عبارت Postfix ازیک عبارت Infix باید تقدم عمگرها را در نظر داشت. برای تبدیل infix به postfix: عبارت براساس تقدم عمگرها پرانتزگذاری شده، سپس از پرانتزهای داخلی شروع و موقعیت هر عملگر به پرانتز بسته مرتبط جا به جا می شود.

Infix -> Postfix

A+B*C

(A+(B*C))

(A+(BC*))

(A (BC*) +)

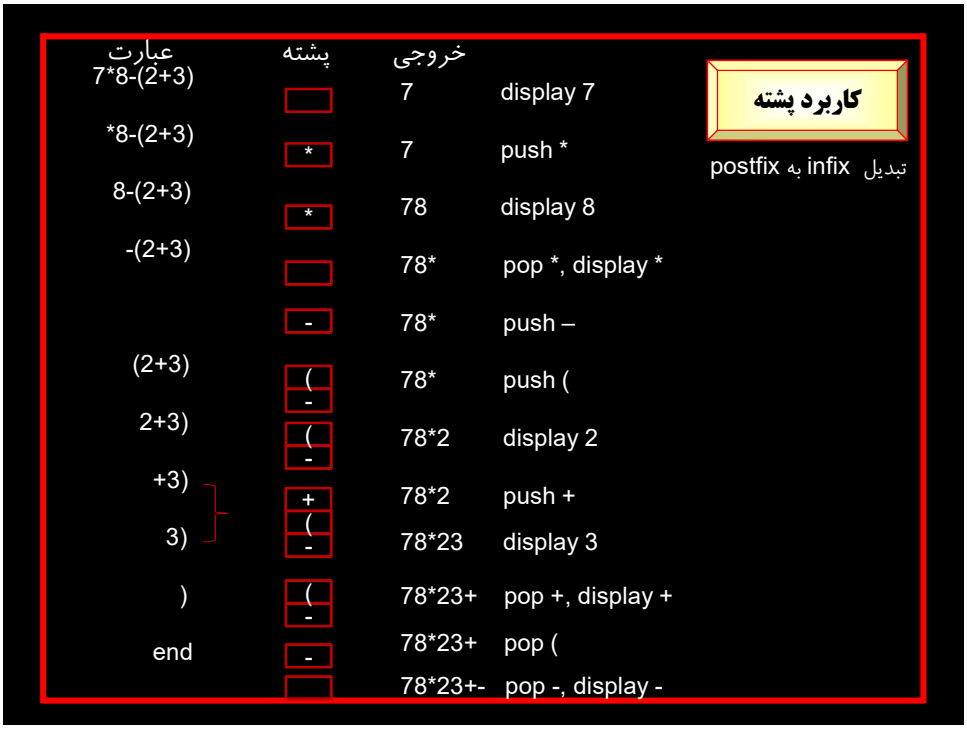
-> A BC* +

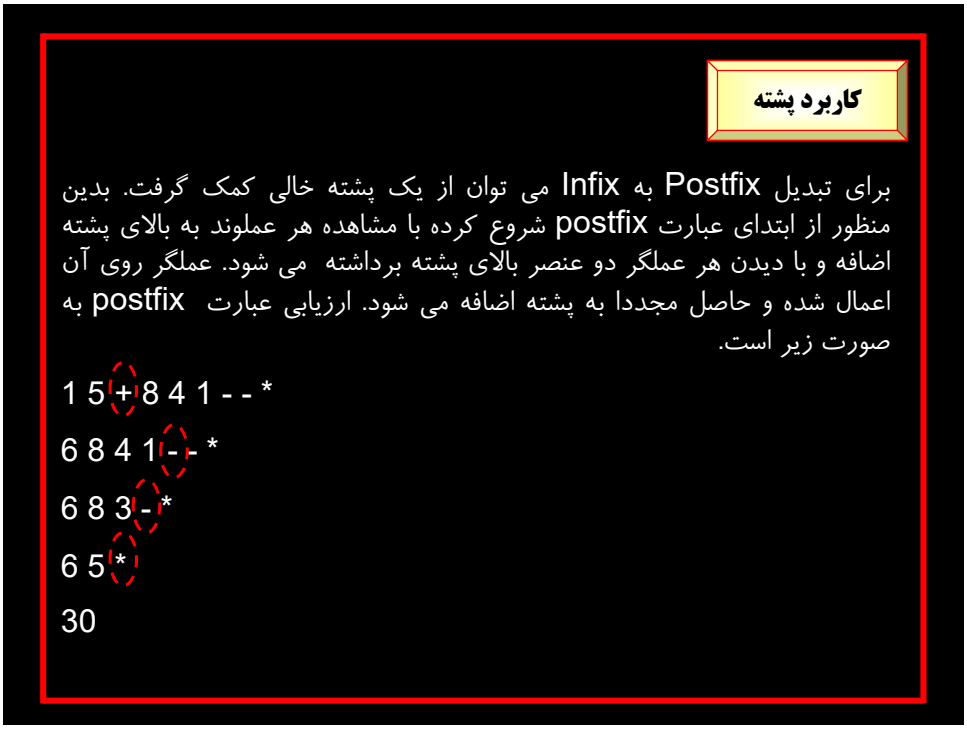
برای تبدیل infix به prefix: عبارت براساس تقدم عمگرها پرانتزگذاری شده، سپس از پرانتزهای داخلی شروع و موقعیت هر عملگر به پرانتز باز مرتبط جا به جا می شود.

کاربرد پشته

تقدم عملگرها

۱	()
۲	sizeof – ++ ~ !
۳	* / %
۴	- +
۵	<< >>
۶	=< < => >
۷	=! ==
۸	&
۹	^
۱۰	|
۱۱	&&
۱۲	||
۱۳	?
۱۴	= =+ =- =* =/ =٪
۱۵	,


کاربرد پشته
تبدیل infix به postfix
عبارت
پشته
خروجی
7*8-(2+3)
7
display 7
*8-(2+3)
*
7
push *
8-(2+3)
*
78
display 8
-(2+3)
78*
pop *, display *
-
78*
push –
(2+3)
(
-
78*
push (
2+3)
(
-
78*2
display 2
+3)
+
78*2
push +
3)
(
-
78*23
display 3
)
(
-
78*23+
pop +, display +
end
-
78*23+
pop (
78*23+-
pop -, display -


کاربرد پشته
برای تبدیل Postfix به Infix می توان از یک پشته خالی کمک گرفت. بدین منظور از ابتدای عبارت postfix شروع کرده با مشاهده هر عملوند به بالای پشته اضافه و با دیدن هر عملگر دو عنصر بالای پشته برداشته می شود. عملگر روی آن اعمال شده و حاصل مجددا به پشته اضافه می شود. ارزیابی عبارت postfix به صورت زیر است.
1 5 + 8 4 1 - - *
6 8 4 1 - - *
6 8 3 - *
6 5 *
30

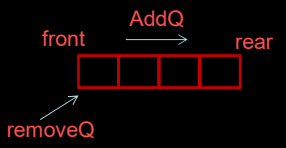
کاربرد پشته

ارزیابی عبارت postfix با پشته

عبارت	پشته
2 4 * 9 5 + -	2
4 * 9 5 + -	4 2
* 9 5 + -	8
9 5 + -	9 8
5 + -	5 9 8
+ -	14 8
-	-6

صف Queue

صف به معنی خط انتظار است. عناصر به ترتیبی که وارد می شوند خدمات میگیرند، یعنی اولین عنصر صف اولین خدمت را میگیرد. صف FIFO (First In First Out) است بر خلاف پشته که LIFO است.
در صف اعمال درج و حذف در دو انتهای آن انجام می شود برخلاف پشته که عمل push و pop در یک طرف لیست انجام می شود.
در صف، عناصر از جلوی صف (front) حذف می شوند و از انتهای صف (rear) به صف اضافه می شوند.



اعمالی که روی صف می توان انجام داد:
۱- افزودن عنصرجدید به انتهای صف (AddQ)
۲- بازیابی عنصری از جلوی صف (Front)
۳- حذف عنصری از جلوی صف (RemoveQ)

صف Queue

برای انجام عمل درج در صف (AddQ) معمولا دو پارامتر نیاز است: صفی که قرار است در آن عنصر جدید درج شود، عنصر جدید که قرار است وارد صف شود.
برای انجام عمل حذف از صف (Remove) یک پارامتر نیاز است آن هم نام صف است. Remove عنصر اول را حذف می کند.
صف نیز مانند پشته فضایی نامحدود تصور می شود که در عمل به علت نامحدود بودن فضا این گونه نیست. بنابراین، باید در نظر گرفت که تعداد اقلام درج شده در صف باعث overflow نشود همچنین واضح است که از یک صف خالی نمی توان حذف کرد چون منجـــر به under flow می شود.
صف ها برای زمانبندی منابع در سیستم کامپیوتری مورد استفاده قرار می گیرند. مثل صف زمانبندی برای خطوط عناصری که باید چاپ شوند، بافرهای ورودی و خروجی، صف های ماندگار کارهایی که منظرند تا از دیسک به حافظه بار شوند، صف آماده از کارهایی که منتظر پردازنده مرکزیند، صف معوق از کارهایی که اجرای آنها به تعویق افتاده.

صف Queue

معایب این روش:
در این روش اعضا به ترتیب وارد صف می شوند و به ترتیب از انتهای صف حذف می شوند ولی فضای عناصری که حذف می شوند دوباره استفاده نمی شوند و صف بعد از مدتی دیگر فضای خالی نخواهد داشت.
دو روش برای حل این معضل وجود دارد:

- شیفت دادن عناصر

✓ اضافه کردن عنصر جدید نیاز به شیفت دادن عناصر به ابتدای صف دارد

- استفاده از صف دایره ای (حلقوی)

بیست مرتبه دو عدد تصادفی تولید شده و در صف قرار می گیرد. از کاربر جمع اعداد پرسیده می شود اگر اشتباه پاسخ داده شود، مجددا دو عدد در صف (Queue) قرار می گیرند.

```
#include "stdafx.h"
#include <iostream>
#include <stdlib.h>

using namespace std;

struct problem
{
        int A;
        int B;
};

problem Q[20];
int front=0;
int rear=-1;
bool AddQ(problem x);
bool RemoveQ(problem &x);
bool QueueIsEmpty();

int main()
{
        problem p;
        int Ans;
        for(int i=0; i<20; i++)
        {
                p.A=rand()%10;
                p.B=rand()%10;
                AddQ(p);
        }
        while (!QueueIsEmpty())
        {
                RemoveQ(p);
                cout<<"First No.:"<<p.A;
                cout<<"\nSecond No.:"<<p.B<<"\n";
                cin>>Ans;
                if(Ans!=p.A+p.B)
                        AddQ(p);
        }
        cout<<"congratulation!!!";
        system("pause");
        return 0;
}
```

```
bool AddQ(problem x)
{
        if(front==0 && rear==19)
                return false;
        if (rear==19)
        {
                int i= front;
                while(i<=rear)
                        Q[i-front]=Q[i++];
                rear=rear-front;
                front=0;
        }
        Q[++rear]=x;
        return true;
}

bool RemoveQ(problem &x)
{
        if(rear<front)
                return false;
        x=Q[front++];
        return true;
}
```

```
bool QueueIsEmpty()
{
        if(rear<front)
                return true;
        else
                return false;
}
```

بازگشتی Recursive

تابع می تواند خودش را فراخوانی کند که این پدیده را بازگشتی گویند. بعضی از مساله ها ماهیتا بازگشتی هستند. مزیت استفاده از توابع بازگشتی سادگی برنامه است اما مصرف زیاد حافظه از معایب این روش است. مساله های توان، فاکتوریل، فیبوناچی از جمله مثالهای رایج برای توابع بازگشتی هستند.

$x^n=x * x^{n-1}$

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

$fibo(n)=fibo(n-1)+fibo(n-2)$

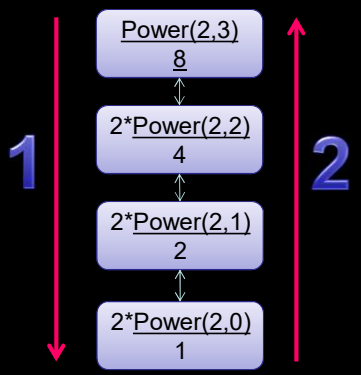
برای محاسبه توابع بازگشتی از پشته استفاده می شود. تابع بازگشتی از دو بخش تشکیل شده است:

۱) حالت پایه (شرط پایان تکرار)

۲) مرحله بازگشتی (فراخوانی تابع توسط خودش)

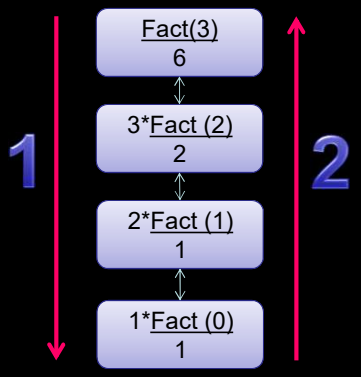
تابع بازگشتی(Recursive) توان

```
#include "stdafx.h"
#include <iostream>
 using namespace std;
 int power(int x, int n);

int main()
{
   int x,n;
   cin>>x>>n;
   cout<<power(x, n);
   system("pause");
   return 0;
}

int power(int x, int n)
{
    if(n==0)
        return 1;
    else
        return x*power(x, n-1);
}
```



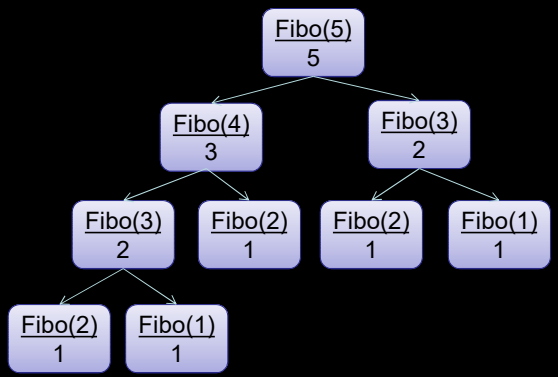
تابع بازگشتی(Recursive) فاکتوریل

```
#include "stdafx.h"
#include <iostream>
 using namespace std;
 int Fact(int x);

int main()
{
        int x;
        cin>>x;
        cout<<Fact(x);
        system("pause");
        return 0;
}

int Fact(int x)
{
        if(x==0)
                return 1;
        else
                return x*Fact(x-1);
}
```



تابع بازگشتی(Recursive) جمله n ام فیبوناچی

```
#include "stdafx.h"
#include <iostream>
 using namespace std;
 int Fibo(int n);

int main()
{
        int n;
        cin>>n;
        cout<<Fibo(n);
        system("pause");
        return 0;
}

int Fibo(int n)
{
        if(n<=2)
                return 1;
        else
                return Fibo(n-1)+Fibo(n-2);
}
```



مرتب سازی Sorting

مرتب سازی یعنی یک لیست (آرایه) را که احتمالاً نامرتب است به شکل صعودی(یا نزولی) مرتب شود. برای مرتب سازی لیستی که تقریبا از قبل مرتب است نسبت به لیستی که اصلا مرتب نیست زمان کمتری لازم است. بنابراین T را می توان در بدترین حالت و بهترین حالت اندازه گرفت. می توان میانگین T را در تمام حالتهای ممکن اندازه گیری کرد(میانگین). در مرتب سازی معمولا بدترین حالت محاسبه می شود. مرتب سازی می تواند inplace (درجا): به فضای اضافی برای مرتب سازی نیاز ندارد یا outplace: به فضای اضافی برای مرتب سازی نیاز دارد، باشد.

انواع مرتب سازی عبارتند از:

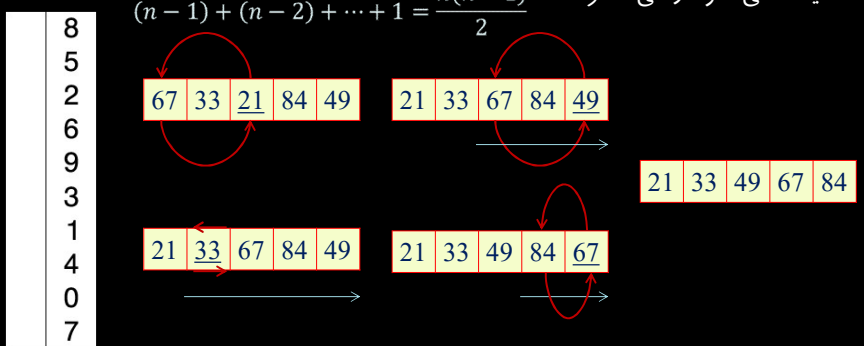
- (Selection sort) مرتب سازی انتخابی
- (Exchange sort) مرتب سازی تعویضی
  - (Bubble sort) مرتب سازی حبابی
  - (Quick sort) مرتب سازی سریع
- (Insertion sort) مرتب سازی درجی
- (Merge sort) مرتب سازی ادغامی
- (Heap sort) مرتب سازی هرمی: یک روش مرتب سازی انتخابی(Selection sort) است.
- ....
- (Radix sort) مرتب سازی مبنایی : مرتب سازی غیر مقایسه ای

مرتب سازی Sorting

Selection *inplace* *Best, Average, Worst* $O(n^2), O(n^2), O(n^2)$

در هر گذر از بخشی از لیست، کوچکترین عنصر آن انتخاب می شود و سپس به جای مناسب خود منتقل می شود یا (بزرگترین عنصر آن انتخاب می شود و سپس به جای مناسب خود منتقل می شود). در گذر اول، اولین عنصر با هر یک از n-1 عنصر دیگر مقایسه می شود. در گذر دوم، دومین عنصر با هر یک از n-2 عنصر دیگر مقایسه می شود و الی آخر.

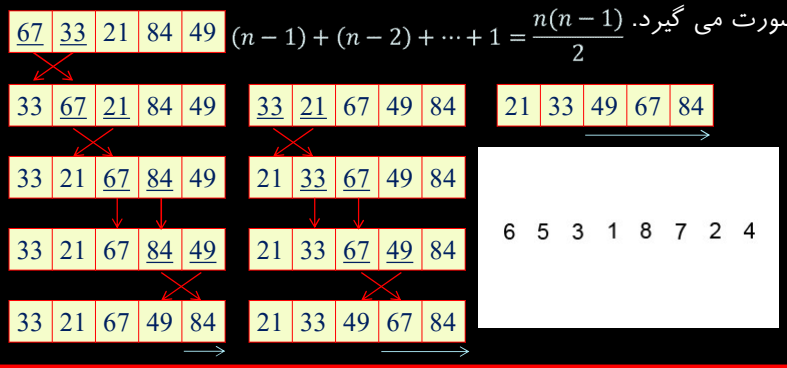
$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$$



مرتب سازی Sorting

Exchange bubble *inplace* *Best, Average, Worst* $O(n), O(n^2), O(n^2)$

جفتهایی از عناصر را که به ترتیب نیستند تعویض می کنند تا چنین جفتهایی وجود نداشته باشند. در مرتب سازی حبابی در هر گذر تضمین می شود بزرگترین عنصر به انتهای لیست می رود و بعضی از عناصر کوچک به سمت ابتدای لیست حرکت می کنند. بدترین حالت زمانی است که عناصر لیست به صورت معکوس باشند. در گذر اول n-1 عنصر مقایسه و تعویض می شوند. در گذر بعدی n-2 تعویض صورت می گیرد. $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$



مرتب سازی Sorting

Exchange quick *inplace* *Best, Average, Worst* $O(n\log n), O(n\log n), O(n^2)$

از استراتژی divide و conquer استفاده می کند. یک روش بازگشتی است. مسئله ای که باید حل شود به مسائل کوچکتر تقسیم می شود و هر کدام مستقلا حل می شود. عنصری به نام محور (pivot) انتخاب می شود و سپس تعویض ها به صورتی انجام می گیرند که عناصر کوچکتر از pivot در سمت چپ و عناصر بزرگتر در سمت راست قرار گیرند. روی هر کدام از این دو لیست مجددا عمل فوق انجام می گردد. اگر عنصر محور میانه آرایه باشد بهترین حالت و اگر کوچکترین یا بزرگترین عنصر باشد، بدترین حالت است.

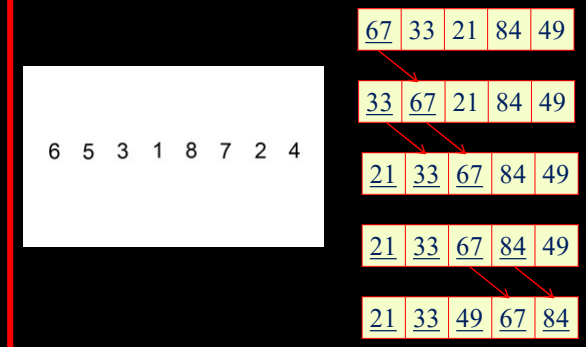
مرتب سازی Sorting

Insertion *inplace* *Best, Average, Worst* $O(n), O(n^2), O(n^2)$

عناصر در لیست مرتب طوری درج می شوند که ترتیب لیست حفظ شوند. بدترین حالت در مرتب سازی درج زمانی است که عناصر لیست به صورت معکوس مرتب باشند.

$2+3+\cdots+n=\frac{n(n-1)}{2}$ -1

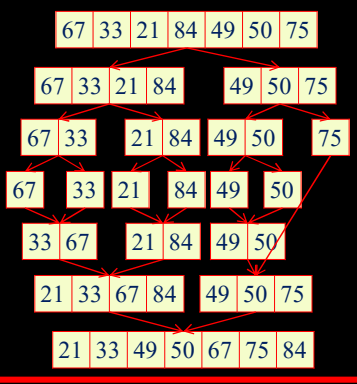
6 5 3 1 8 7 2 4



مرتب سازی Sorting

Merge *outplace* *Best, Average, Worst* $O(n^1ogn), O(n^1ogn), O(n^1ogn)$

6 5 3 1 8 7 2 4



روش Merge sort به نوع مقادیر ورودی وابستگی ندارد و از استراتژی divide و conquer استفاده می کند. در اکثر پیاده‌سازی‌ها این الگوریتم پایدار(stable) می‌باشد. بدین معنی که این الگوریتم ترتیب ورودی‌های مساوی را در خروجی مرتب شده حفظ می‌کند. پیچیدگی Merge sort تا لحظه ای که لیست ها تک عنصری می شوند برابر با (nlogn) می شود از طرفی اعمال ترکیب تا رسیدن به لیست واحد نیز انعکاسی از بخش فوقانی است پس پیچیدگی آن نیز (nlogn) خواهد بود. پیچیدگی کل همان (2nlogn) است که برابر با (nlogn) است.

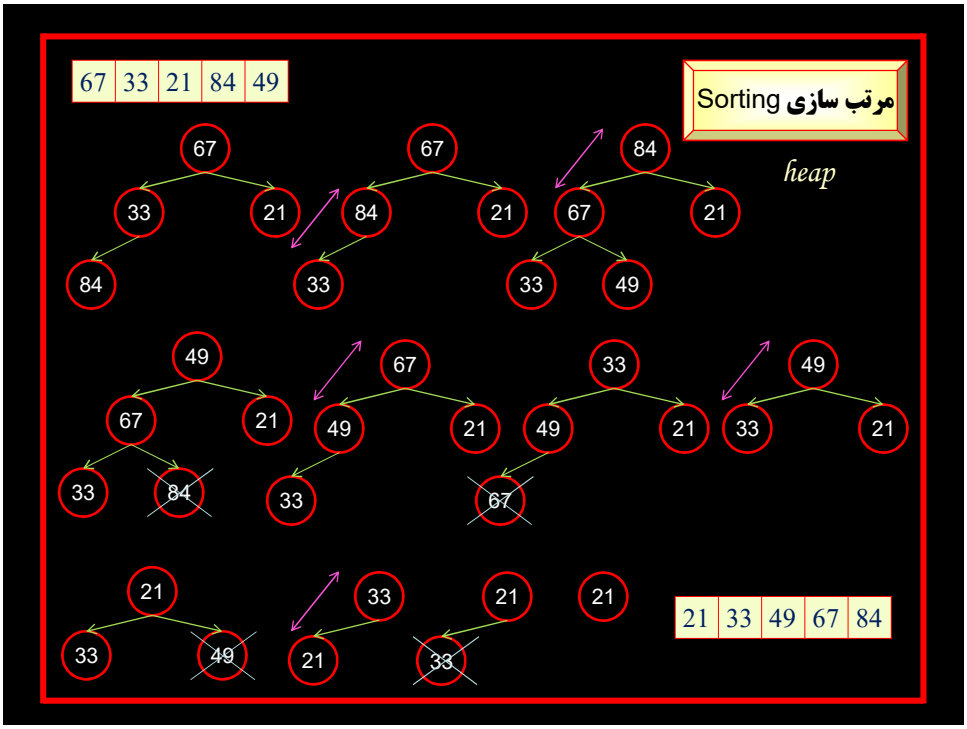
مرتب سازی Sorting

Selection *inplace* *Best, Average, Worst*
heap *O(nlogn), O(nlogn), O(nlogn)*

هرم یک درخت دودویی با ویژگیهای زیر است: کامل است (هر سطح درخت کاملا پر است، احتمالا به جز سطح پایین که گره ها در این سطح در سمت چپ ترین موقعیت قرار دارند)، ویژگی ترتیب هرم را دارد (داده ذخیره شده در هر گره بزرگتر یا مساوی داده های موجود در فرزندانش است: maxheap). فرزندان گره i در 2*i و 2*i+1 و پدر آن در i/2 قرار دارد. ابتدا درخت به هرم تبدیل می شود. سپس بزرگترین عنصر با آخرین فرزند جابجا شده و درخت هرس می شود و این عمل تکرار می شود تا تمام گره ها هرس شوند. در این میان اگر درخت از حالت هرم خارج شد مجددا به هرم تبدیل می شود.

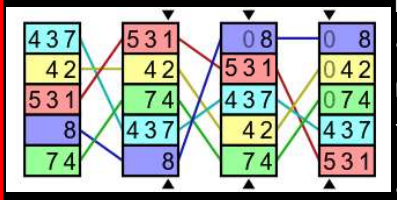
6 5 3 1 8 7 2 4

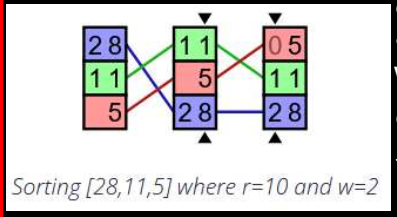
هرم با عمل درج و حذف نامتوازن نمی شود اماBST (در آینده ارائه خواهد شد) با حذف و درج نامتوازن می شود. هرم بهترین DS برای پیاده سازی صف اولویت است.



مرتب سازی Sorting

Radix *outplace* *Best, Average, Worst*
O(wn), O(wn), O(wn)





Sorting [28,11,5] where r=10 and w=2

مرتب‌سازی مبنایی الگوریتمی است که لیست n عضوی به طول w را در زمان O(wn) مرتب می کند. ورودی‌ها به بخش‌های کوچکی تقسیم می‌شوند (اگر یک کلمه است به حرف‌هایش و اگر عدد است به ارقامش شکسته می شود).
ابتدا لیست بر اساس کم ارزش‌ترین بیت (حرف یا رقم) مرتب می‌شود، سپس بر اساس دومین بیت، تا در نهایت بر اساس پرارزش‌ترین بیت مرتب سازی انجام می شود(LSD). پس از w مرحله لیست مرتب می‌شود. این روش مرتب‌سازی پایدار است و در تهیه واژه نامه‌ها و مرتب‌سازی اعداد استفاده می‌شود.
پیچیدگی مکانی O(n+r) است که r مبنا است.
مرتب سازی غیر مقایسه ای است.

جستجو Searching

جستجوی عنصری در یک آرایه مرتب:

- جستجوی خطی

```
#include "stdafx.h"
#include <iostream>
using namespace std;
int LinearSearch(int A[], int x);
int main()
{
        int x;
        int A[]={3,6,9,10,14,18,19,20,25,30};
        cin>>x;
        cout<< LinearSearch( A,  x);
        system ("pause");
        return 0;
}
int LinearSearch(int A[], int x)
{
        int i;
        for(i=0;i<10;i++)
        {
                if(A[i]==x)
                {
                  cout<<"found in index:";
                  return i;
                }
        }
        return -1;
}
```

اگر عنصر x در آرایه وجود دارد
اندیس محل عنصر به برنامه برگردانده می شود

جستجو Searching

جستجوی عنصری در یک آرایه مرتب:

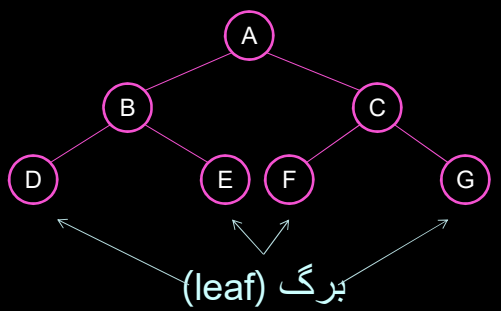
- جستجوی باینری

```
int BinarySearch(int A[], int x,int L, int F);
int main()
{
        int x,last,first;
        int A[]={3,6,9,10,14,18,19,20,25,30};
        cin>>x;
        last=9; first=0;
        cout<< BinarySearch( A, x, last, first);
        system ("pause");
        return 0;
}
int BinarySearch(int A[], int x,int L, int F)
{
        int i;
        i=(L+F)/2;
        if(F>L)
                return -1;
        else if(A[i]>x)
                BinarySearch( A,  x, i-1,  F);
        else if(A[i]<x)
                BinarySearch( A,  x, L,  i+1);
        else if(A[i]==x)
                return i;
}
```

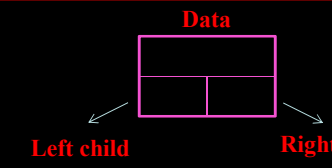
x با عنصر وسط لیست Aمقایسه می شود.
اگر برابر باشد اندیس عنصر برگردانده می شود
اگر بزرگتر باشد نیمه راست لیست
اگر کوچکتر باشد نیمه چپ لیست
بررسی می شود.

درخت Tree

هر عنصر درخت گره (node) نامیده می شود. درختهایی که در آنها گره ها حداکثر دارای دو فرزند باشند درختهای دودویی می نامند. گره هایی که یک پدر دارند همزاد (sibling) می نامند. گره های آخرین سطح برگ (leaf) نامیده می شوند. تعداد زیر درخت های هر گره درجه (degree) آن گره است. فاصله هر گره تا ریشه درخت را سطح آن گره می نامند. بزرگترین درجه گره در درخت، درجه درخت نامیده می شود.



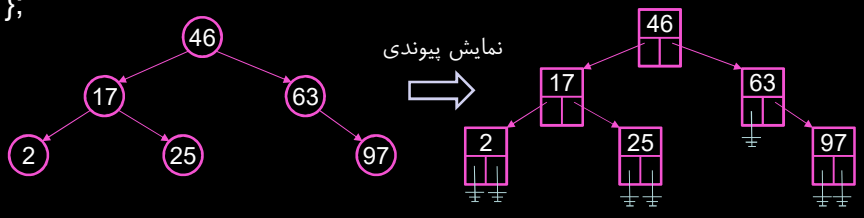
درخت Tree



نمایش پیوندی درختهای دودویی

اگر اشاره گری به ریشه درخت اشاره کند، از طریق آن می توانیم به تمام گره های درخت دستیابی داشته باشیم.

```
struct  Tree {
Int data;
Tree *left;
Tree *right;
};
```

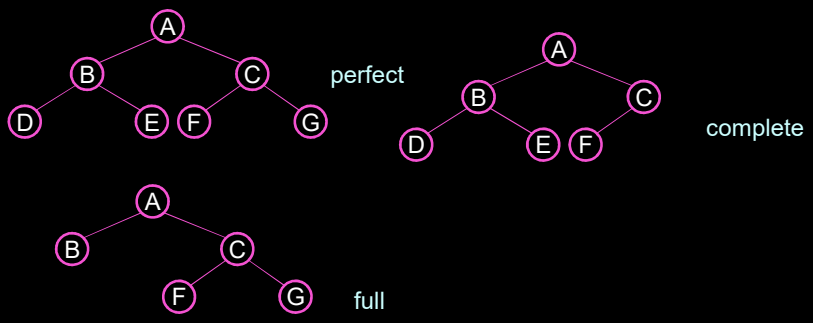


درخت Tree

- درختی که حداکثر تعداد فرزندان هر گره آن برابر k باشد را درخت k تایی می نامند. معروفترین درخت k تایی درخت دودویی است.
- درخت کاملاً پر (Perfect): درختی است که همه گره ها بجز گره های سطح آخر (برگها) دارای حداکثر فرزندان بوده (حداکثر درجه درخت) و برگها هم سطح باشند.
- درخت کامل (Complete) : درختی است که اگر ارتفاع آن d باشد تا ارتفاع d −۱ درخت Perfect بوده و برگها در سطح d تا حد ممکن در سمت چپ باشند.
- درخت پر (Full): درختی که در آن گره ها یا برگ هستند و یا به تعداد درجه درخت فرزند دارند را درخت fullگویند.
- حداکثر تعداد گره ها در یک درخت دودویی به ارتفاع d برابر با $2^{d+1}-1$ است.
- حداکثر تعداد گره ها در یک درخت دودویی با تعداد سطح l برابر با $۲^{l}-۱$ است.
- درختی دودویی با n گره دارای ارتفاع $\log^{n+1}-1$ است و می توان درخت را n! حالت خواند و تعداد سطح برابر است با $\log^{n+1}$.
- تعداد نودها (گره ها) برابر است با $n=2^0 + 2^1 + \ldots + 2^L$ (L: پایین ترین سطح)
- در درخت دودویی کامل با n گره، فرزندان گره i در 2*i و 2*i+1، پدر در i/2 قرار دارد.
- درخت دودویی کاملی که مقدار کلید هر گره کوچکتر یا مساوی با گره های فرزندان باشد (minheap)- درخت دودویی کاملی که مقدار کلید هر گره بزرگتر یا مساوی با گره های فرزندان باشد (maxheap)

درخت Tree

- درختی که هر گره آن یا فرزند ندارد یا دو فرزند دا د درخت دودویی محض نامیده می شود.
- درخت perfect هم Full و هم complete است.



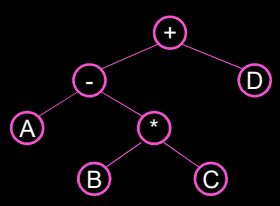
درخت Tree

دو روش کلی برای پیمایش درخت (حرکت در طول درخت و ملاقات تمام گره ها (هر گره فقط یک بار ملاقات می شود)) وجود دارد:

- ✓ (BFS): پیمایش سطحی: درخت سطح به سطح (از بالا به پایین) و از چپ به راست پیمایش میشود.(صف)
- ✓ (DFS): پیمایش عمقی: در این شیوه، پیمایش درخت تا انتهای یک عمق پیشروی می کند تا به انتها برسد سپس به عقب باز میگردد تا به اولین مسیر جدید برسد.(پشته)

پیمایش عمقی به سه روش انجام می گیرد: (L:left, N:node, R:right)

(LNR) Inorder (NLR) Preorder (LRN) Postorder

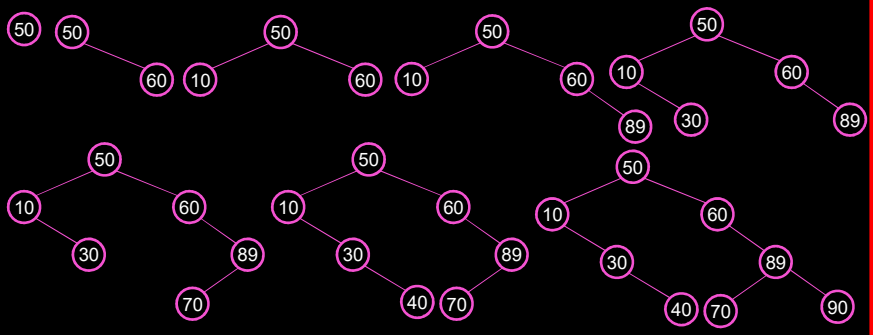


DFS
- Inorder: A-B*C+D
- Preorder: +-A*BCD
- Postorder: ABC*-D+

BFS
- +-DA*BC

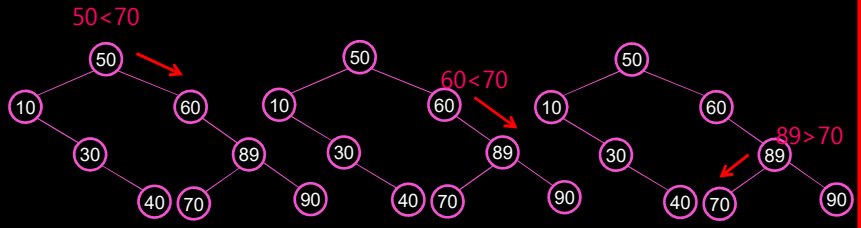
درخت Tree

درخت BST دارای ویژگیهای زیر است: مقدار موجود در هر گره بزرگتر از فرزند چپ و کوچکتر از مقدار فرزند راست (در صورت وجود) است. زمان جستجو و درج در درخت BST به صورت *O(log n)* است. زمان ایجاد درخت *O(n log n)* است.
درخت BST داده های زیر را ایجاد کنید. ۵۰,۶۰,۱۰,۸۹,۳۰,۷۰,۴۰,۹۰



درخت Tree

جستجوی درخت BST: فرض کنید خواسته باشیم دنبال عنصری با کلیدkey بگردیم. ابتدا از ریشه شروع می شود. اگر ریشه تهی باشد درخت جستجو فاقد هر عنصری بوده و جستجو ناموفق خواهد بود. درغیراین صورت key با مقدار کلید ریشه مقایسه می شود، اگر key برابر مقدار کلید ریشه باشد، جستجو موفق است. اگر key کمتر از مقدار کلید ریشه باشد، زیردرخت چپ ریشه جستجو می شود. اگر keyبزرگتر از مقدار کلید ریشه باشد، زیردرخت راست ریشه جستجو می شود.
جستجوی عدد ۷۰ در درخت زیر:



درخت BST یکتا نیست. BST بر اساس ترتیب درج داده ها متفاوت خواهد بود.

درخت Tree

کد هافمن: الگوریتم فشرده سازی هافمن را دیوید هافمن پروفسور دانشگاه MIT آمریکا اختراع کرد. روش فشرده سازی هافمن الگوریتمی است که برای فشرده سازی متن مناسب می باشد.

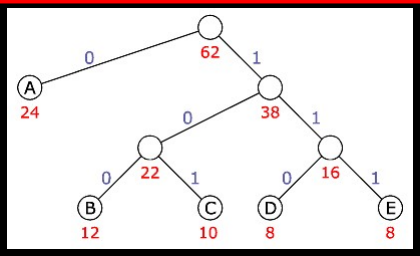
الگوریتم هافمن جزو خانواده الگوریتمهایی است که طول کد متغیری دارند. این به آن معناست که نمادهای مجزا (برای نمونه کاراکترهایی در یک فایل متنی) با رشته بیتهایی که طولهای مختلفی دارند تعویض میشود.

بنابراین نمادهایی که زیاد در یک فایل تکرار می شوند یک رشته بیت کوتاه می گیرند در حالی که نمادهای دیگر که به ندرت دیده می شوند رشته بیت طولانی تری را میگیرند.

درخت هافمن به (strict محض) تبدیل می شود.

درخت Tree

کاراکتر	تعداد تکرار
A	24
B	12
C	10
D	8
E	8



کاراکتر	تعدادتکرار	کد	طول کد	کل طول
A	24	0	1	24
B	12	100	3	36
C	10	101	3	30
D	8	110	3	24
E	8	111	3	24

رمزنگاری هافمن

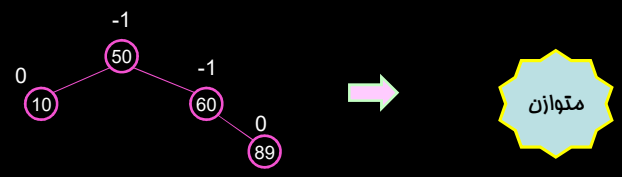
رمزگشایی هافمن

1000110

B A D

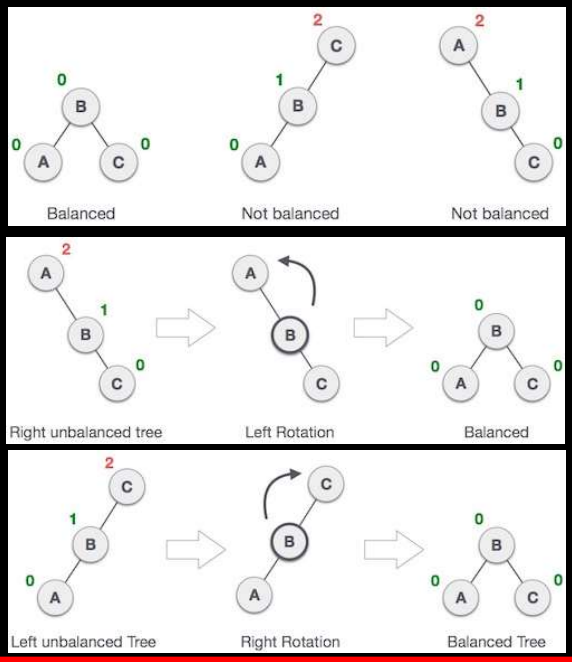
درخت Tree

درخت متوازن: درختی است که حداکثر اختلالف ارتفاع دو زیر درخت چپ و راست در هر گره آن یک می باشد. درخت BST متوازن را درخت AVL گویند.
ضریب توازن برابر است با ارتفاع زیردرخت چپ – ارتفاع زیردرخت راست.
ضریب توازن در درخت AVL ۰، ۱ یا ۱- است.
مرتبه اجرایی هر عمل درج، حذف، جستجو o(logn) است.



درخت Tree

درخت متوازن (AVL) : درج



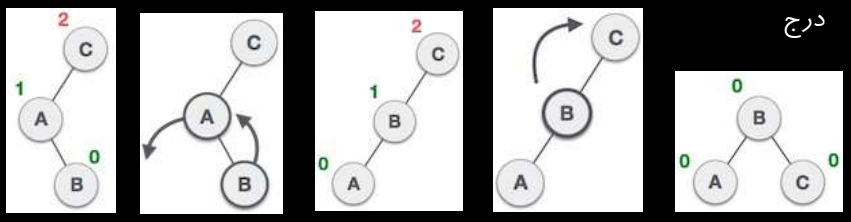
L

R

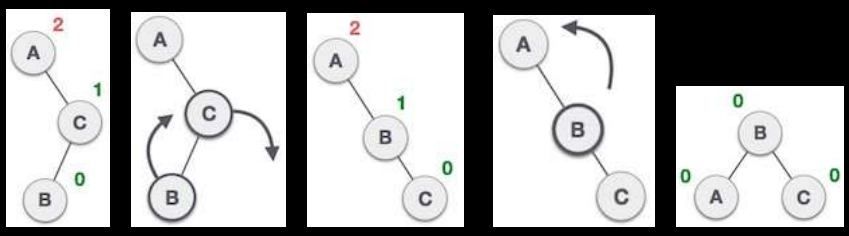
درخت Tree

درخت متوازن (AVL) : درج

LR

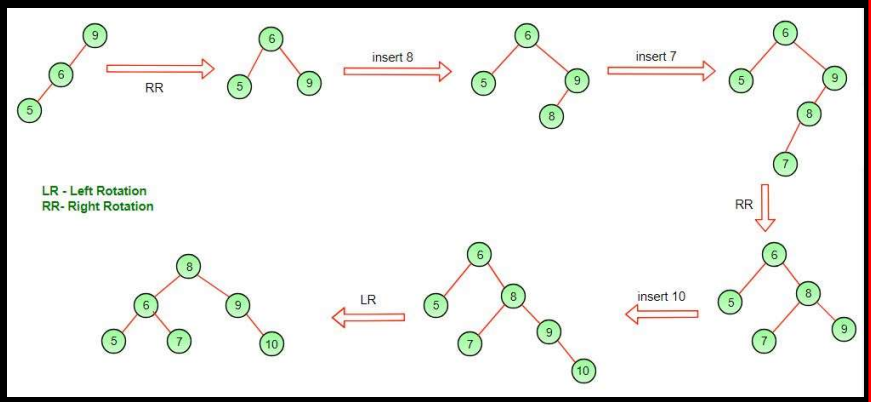


RL



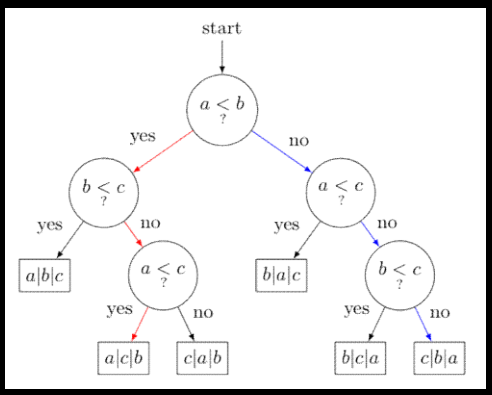
درخت Tree

درخت متوازن (AVL) : مثال درج



درخت Tree

درخت تصمیم (Decision Tree): درخت تصمیم مانند یک درخت است با این تفاوت که از ریشه به سمت پایین (برگ) رشد کرده است. در هنگام ساخت درخت، ابتدا ریشه ساخته می‌شود، سپس هر یک از زیر شاخه ها به شاخه‌های دیگری تقسیم می‌شود و این فرآیند تکرار می شود.



- ✓ تعداد مقایسه های انجام شده توسط درخت در بدترین حالت برابر با عمق درخت است.
- ✓ درخت تصمیم گیری که فقط از طریق مقایسه کردن عمل مرتب سازی را انجام می دهد، دارای n! برگ است.
- ✓ در بدترین حالت log n! مقایسه انجام می شود.

درخت Tree

درخت قرمز و سیاه (درخت متوازن):

برای رفع مشکل عدم توازن درخت BST درخت قرمز و سیاه ارائه شده است.

درخت قرمز سیاه، یک درخت دودویی جست و جو است که دارای ویژگی های زیر است.

۱- هر گره این درخت، به رنگ قرمز یا سیاه است.
۲- ریشه درخت، سیاه است.
۳- رنگ هر برگ (NIL) سیاه است.
۴- پدر هر گره قرمز، سیاه است.
۵- برای هر گره، هر مسیر ساده از آن گره تا هر برگ، تعداد گره سیاه یکسان دارد(ارتفاع سیاه).

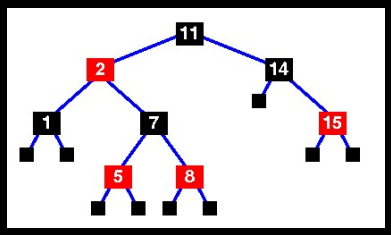
حداکثر ارتفاع یک درخت قرمز-سیاه که دارای n گره داخلی باشد $2\log(n+1)$

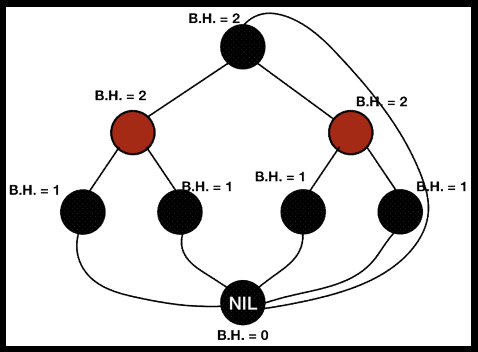
درج در درخت قرمز و سیاه:

- ابتدا به صورت BST درج انجام می شود.
- رنگ گره قرمز می شود.
- در صورت نیاز درخت اصلاح می شود.

Tree **درخت**

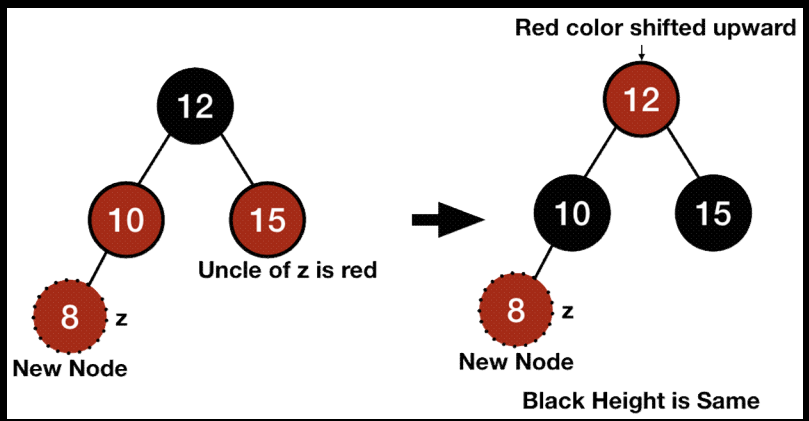
درخت قرمز و سیاه





Tree **درخت**

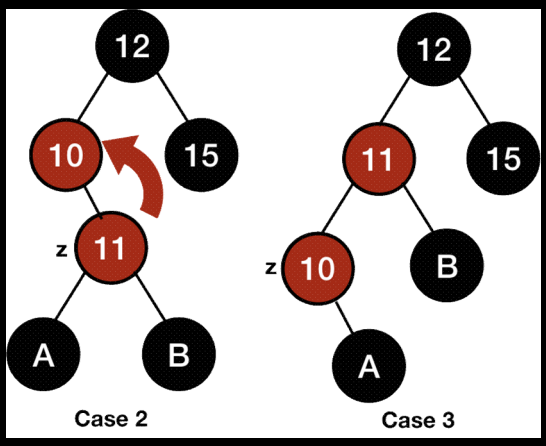
درج در درخت قرمز و سیاه : حالت اول (پدر و عمو قرمز)



درخت Tree

درج در درخت قرمز و سیاه : حالت دوم

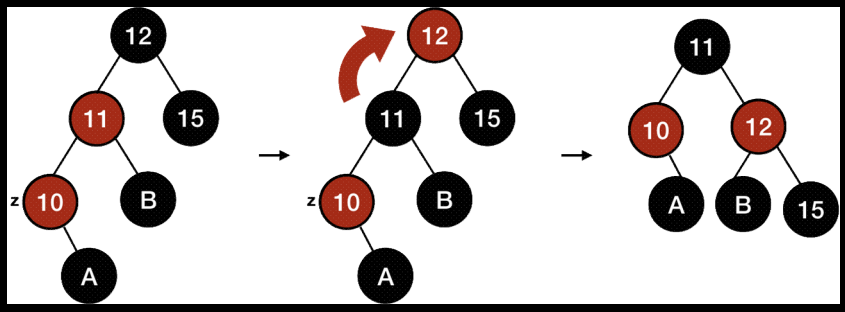
(پدر قرمز عمو سیاه، رابطه با پدر و پدربزرگ غیرخطی)



درخت Tree

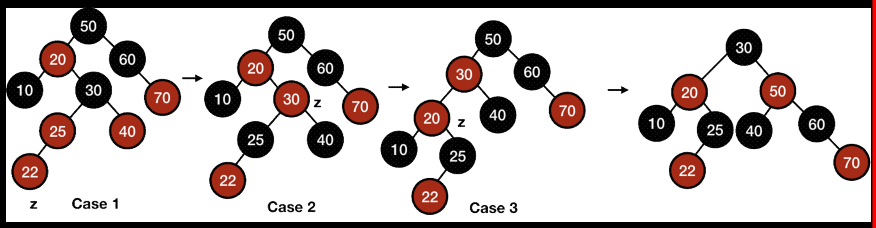
درج در درخت قرمز و سیاه : حالت سوم

(پدر قرمز عمو سیاه، رابطه با پدر و پدربزرگ خطی)



درخت Tree

مثال درج

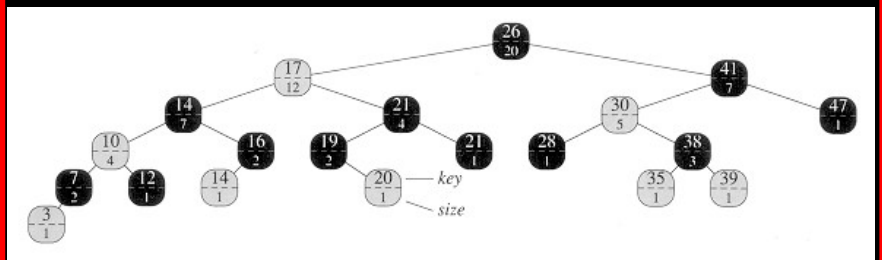


درخت Tree

درخت مرتبه آماری: یک درخت قرمز و سیاه است که n داده را طوری سازماندهی می کند که اعمال مرتبه آماری (یافتن مرتبه یک عنصر، یافتن عنصری با مرتبه خاص) در O(logn) انجام می شود.

هر گره در درخت مرتبه آماری علاوه بر اطلاعات درخت قرمز و سیاه، یک مولفه size[x] هم داردکه تعداد عناصر موجود در زیردرختی به ریشه x را نشان می دهد.

size[x]=size[left[x]]+size[right[x]]+1

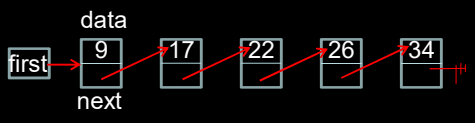


لیست پیوندی Linked list

لیست دنباله متنهاهی از عناصر داده است و عناصر آن دارای ترتیب هستند: اولین عنصر، دومین عنصر و .... لیست پیوندی مجموعه مرتبی از عناصر به نام گره (node) است که هر گره دارای دو بخش است:

۱- بخش داده

۲- بخش آدرس که حاوی اشاره گری است که محل گره ای را که حاوی عنصر بعدی لیست است مشخص می کند. اگر عنصر بعدی موجود نباشد مقدار تهی در آن قرار می گیرد.



پیکانها پیوندها را نشان می دهند. اشاره گر first (در بعضی مراجع head نامیده می شود) به اولین گره لیست اشاره می کند. بخش data مقداری را در لیست ذخیره می نماید و null در بخش پسوند گره آخر لیست، انتهای لیست را نشان می دهد.

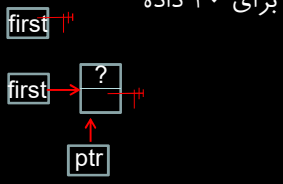
لیست پیوندی Linked list

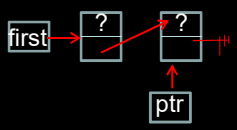
ساختار گره

```
struct Node{
int data;
Node *next;
};
```

ایجاد لیست برای ۲۰ داده

```
int main()
{
        Node *first=NULL;
        Node *ptr;
        first= new (Node);
        first -> next = NULL;
        cin>> first->data;
        ptr=first;
        for (int i=0; i<19;i++)
        {
                ptr-> next=new Node;
                ptr=ptr->next;
                cin>>ptr->data;
                ptr->next=NULL;
        }

        return 0;
}
```





لیست پیوندی Linked list

پیاده سازی عملیات اصلی لیست

- ✓ ایجاد لیست: برای ایجاد لیست خالی، first را برابر null قرار می دهیم تا بیانگر این باشد که به گره ای اشاره نمی کند. first = NULL;
- ✓ تست خالی بودن لیست: if(first==NULL)
- ✓ پیمایش لیست: ptr= first; ptr=ptr->next;
- ✓ درج در لیست: ابتدا گره جدیدی ایجاد کرده سپس در لیست درج می شود. درج در ابتدای لیست، وسط لیست و انتهای لیست
- ✓ حذف از لیست: حذف عنصر از ابتدای لیست، حذف عنصر از وسط لیست، حذف عنصر از انتهای لیست

در پردازش لیست های پیوندی به حالتهای خاص باید توجه داشت:

- اگر سعی شود عناصر خارج از لیست پردازش شوند، خطا اتفاق افتاده است.
- از لیست خالی نمی توان به عنصری دستیابی داشته باشید.
- در هنگام عملیات، پیوند بین گره ها به درستی تغییر یابند.

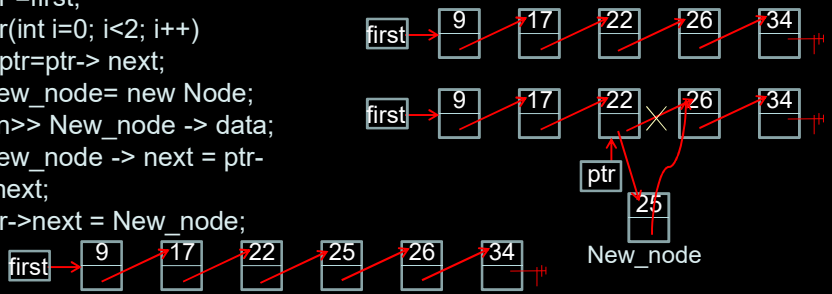
لیست پیوندی Linked list

درج در ابتدا:

```
Node * New_node;
New_node= new Node;
cin>> New_node -> data;
New_node -> next = first;
first= New_node;
```

درج در وسط:

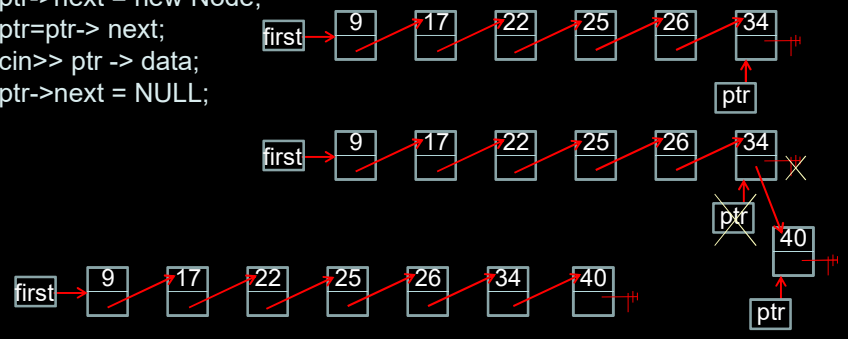
```
Node * New_node;
ptr =first;
for(int i=0; i<2; i++)
  ptr=ptr-> next;
New_node= new Node;
cin>> New_node -> data;
New_node -> next = ptr->next;
ptr->next = New_node;
```



لیست پیوندی Linked list

درج در انتها:

```
ptr =first;
while ( ptr->next != NULL)
      ptr=ptr-> next;
ptr->next = new Node;
ptr=ptr-> next;
cin>> ptr -> data;
ptr->next = NULL;
```



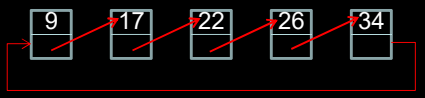

```
struct Node{
int data;
Node *next;
};
int main()
{
  Node *first=NULL;
  Node *ptr, *ptr1;
  first= new (Node);
  first -> next = NULL;
  cin>> first->data;
  ptr=first;
  for (int i=0; i<4;i++)
   {
            ptr->next=new Node;
            ptr=ptr->next;
            cin>>ptr->data;
            ptr->next=NULL;
   }
  int delData;
  cin>>delData;
  if(first->data == delData)
   {
            ptr = first->next;
            delete first;
            first = ptr;
   }
      else
      {
          ptr=first;
          ptr1=first->next;
          while (ptr->next != NULL && ptr1->data != delData)
              {
                    ptr1 = ptr1->next;
                    ptr = ptr->next;
               }
                    if(ptr->next == NULL && ptr1->data != delData)
                        cout<<"Not found";
                    else
                    {
                        ptr->next=ptr1->next;
                        delete ptr1;
                    }
      }

      ptr=first;
      while (ptr != NULL )
      {
            cout<< ptr->data<<"\t";
            ptr = ptr->next;
      }
      system ("pause");
      return 0;
}
```

حذف عددی(delData) از لیست

لیست پیوندی Linked list

Link list حلقوی: بخش آدرس آخرین گره به اولین گره اشاره می کند.

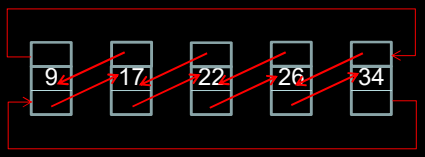
لینک لیست یک طرفه



لینک لیست یک طرفه حلقوی

لینک لیست دو طرفه

```
struct Node{
int data;
Node *next;
Node *prev;
};
```

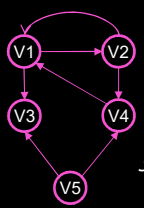


لینک لیست دو طرفه حلقوی

گراف Graph

درخت حالت خاصی از یک ساختار کلی تر به نام گراف جهت دار است. تفاوت درختها و گرافهای جهت دار این است که گراف جهت دار فاقد گره ریشه است و ممکن است چند مسیر (یا هیچ مسیری) از یک گره به گره دیگر وجود داشته باشد. گراف حاوی تعداد متناهی از عناصر به نام گره (Vertex) است که جفتهایی از گره ها که با تعداد متناهی از کمان ها یا لبه (Edge) به هم وصل شده اند.

V(G1)={V1,V2,V3,V4,V5}
E(G1)={<V1,V2>,<V2,V1>,<V1,V3>,<V5,V3>,<V5,V4>,<V4,V1>,<V2,V4>}



اگر n تعداد راس ها باشد:
حداکثر تعداد کل یالها در گراف کامل غیرجهت دار n(n-1)/2
حداکثر تعداد کل یالها در گراف کامل جهت دار n(n-1)
درجه هر گره: تعداد یالهایی که از یک گره عبور می کند
درجه خروجی برای گراف های جهت دار: تعداد یالهای خارج شده از یک گره
درجه ورودی برای گراف های جهت دار» تعداد یالهایی که به یک گره وارد شده اند
گرا ف وزن دار: گرافی است که به هر یال آن وزن (هزینه) داده می شود

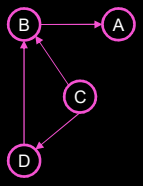
گراف Graph

انواع نمایش گراف

- ماتریس همجواری adjacency matrix
- لیست همجواریadjacency list

❖ ماتریس همجواری

در گراف جهت دار اگر یال وجود داشته باشد آن گاه A(i,j)=1 درغیراینصورت ۰ خواهد بود. اگر از i به j یال نباشد A[i][j]=0 و اگر از i به j یال باشد A[i][j]=1. ماتریس همجواری گراف های بدون جهت روی قطر اصلی متقارن است اما در گراف های جهت دار ممکن است متقارن نباشد.



	A	B	C	D
A	0	0	0	0
B	1	0	0	0
C	0	1	0	1
D	0	1	0	0

فضای مصرفی ماتریس همجواری $o(|V|^2)$

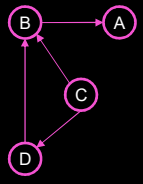
گراف Graph

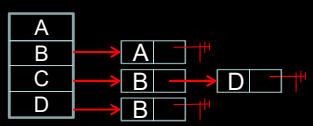
انواع نمایش گراف

- ماتریس همجواری adjacency matrix
- لیست همجواریadjacency list

❖ لیست همجواری

مشکل ماتریس همجواری این است که این ماتریس اغلب اسپارس است به این معنی که بسیاری از عناصر آن صفر است و نیاز به مقدار زیادی حافظه دارد. به همین دلیل از لیست همجواری استفاده می شود. فضای مصرفی لیست همجواری $o(|V|+|E|)$ است.





گراف Graph

adj^k تعداد مسیرهای به طول k از گره i به گره j را نشان می دهد. برای بدست آوردن مسیری به طول ۲ ماتریس همجواری به توان ۲ رسانده می شود(ماتریس را در خودش ضرب می شود). برای بدست آوردن مسیری به طول ۳ ماتریس همجواری به توان ۳ رسانده می شود.

مسیرهای به طول دو برای گراف صفحه قبل:

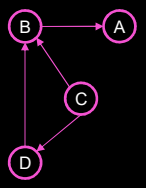
0	0	0	0
1	0	0	0
0	1	0	1
0	1	0	0

×

0	0	0	0
1	0	0	0
0	1	0	1
0	1	0	0

=

	A	B	C	D
A	0	0	0	0
B	0	0	0	0
C	1	1	0	0
D	1	0	0	0



مسیر به طول ۲ از C به A و از C به B و از D به A وجود دارد.

گراف Graph

پیمایش گراف

✓ جستجوی سطحی Breadth- First Search (BFS)

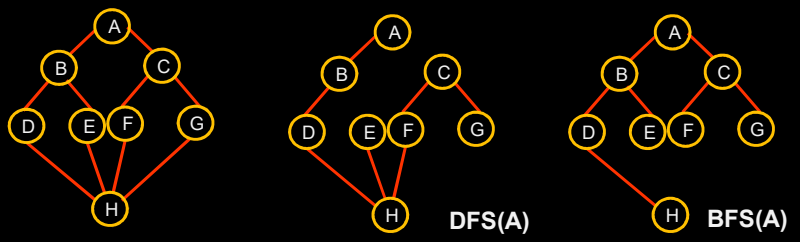
در پیمایش سطحی با شروع از یک گره و ملاقات آن کلیه گره های مجاور آن نیز ملاقات می شوند. سپس این رویه به ترتیب برای هر یک از گره های مجاور تکرار می شود. برای پیاده سازی پیمایش سطحی از ساختمان داده صف استفاده می شود. بدین ترتیب که رئوس مجاور هنگام ملاقات وارد صف می شوند سپس از سر صف یک عنصر را حذف کرده و گره های مجاور آن ضمن ملاقات به صف اضافه می شوند. هر گره در صورتی وارد صف می شود که قبلاً نباشد. درخت پوشای سطحی لزوماً منحصر به فرد نیست.

✓ جستجوی عمقی Depth- First Search (DFS)

در پیمایش عمقی با شروع از یک گره و ملاقات آن، یکی از گره های مجاوری که ملاقات نشده ملاقات می شوند و این عمل متناوباً تکرار می شود. اگر در یک گره، همه گره های مجاور آن ملاقات شده بودند یک مرحله به عقب میرویم. برای پیاده سازی درپیمایش عمقی از پشته استفاده می شود. درخت پوشای عمقی لزوماً منحصر به فرد نیست.

گراف Graph

پیمایش گراف



DFS : A , B , D , H , E , F , C , G
BFS : A , B , C , D , E , F , G , H

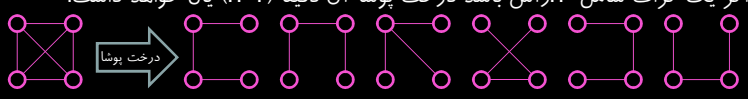
گراف Graph

درخت پوشا Spanning Tree:

درخت پوشای یک گراف درختی است شامل دقیقا تمام رئوس گراف و تعدادی از یالهای آن.
درخت پوشا در یک گراف درختی است که تمام رئوس را پوشانده باشد. (درخت خود یک گراف همبند است با این شرط که سیکل یا دوره نداشته باشد.)

هنگام رسم درخت پوشای گراف با nراس به نکات زیر باید توجه کرد:

- فقط مجاز به استفاده از یالهای موجود در گراف هستیم.
- از رسم یالهایی که ایجاد حلقه می کنند پرهیز می کنیم.
- درخت پوشا منحصر به فرد نیست.
- اگر یک گراف شامل nرأس باشد درخت پوشا آن دقیقاً (n-1) یال خواهد داشت.



درخت پوشای کمینه (minimum spanning tree):

در یک گراف بدون جهت وزن دار، از بین درختان پوشایی که می توان برای چنین گرافی در نظر گرفت درخت یا درختانی وجود دارند که مجموع وزن یالهای آن درختان پوشا minimum باشد.
الگوریتم های مختلفی برای پیدا کردن این درخت پوشای مینیمم وجود دارد که عبارتند از Kruskal، prim، Dijkstra، Floyd.

درهم سازی Hashing

درهم سازی

✓ هزینه

بهترین هزینه درج، حذف و جستجو در ساختمان داده هایی که تا به حال مورد مطالعه قرار داده شدند log n است. یا استفاده از Hashing هزینه این اعمال O(1) خواهد بود.

	لیست نامرتب	لیست مرتب	درخت (هرم / BST)
Insert	1	n	log n
Search	n	log n	log n
Delete	n	n	log n

✓ آدرس دهی مستقیم

در direct address هر عنصر برحسب مقدار کلیدش مستقیم در یک آرایه T ذخیره می شود O(1).

$$T(k) = \begin{cases} x & k \in K, key[x] = k \\ null \end{cases}$$

$$K \subseteq \{0,1,\ldots,m-1\}$$

درهم سازی Hashing

درهم سازی

✓ جداول درهم سازی

- اگر m بسیار بزرگ باشد ساخت یک جدول به بزرگی m ناممکن است (مشکل).
- برای حل این مشکل از جداول درهم سازی استفاده می شود.
- جدول به اندازه قابل پیاده سازی m تعمیم داده می شود.
- عنصر x با کلید k=key[x] در درایه h(k) ذخیره می شود.
- h تابع درهم سازی (hash function) است
- مقدار h(k) به ازای یک کلید k، یک عدد صحیح بین ۰ و m-1 است.

✓ برخورد (collision)

- در جدول درهم سازی امکان دارد بیش از یک عنصر در یک درایه نگاشت شود. به این مشکل collision گفته می شود(به ازای دو کلید متفاوت آدرس یکسانی تولید شود).
- امکان collision در هر تابع hash امکان پذیر است مشکل را با آدرس دهی باز یا روش زنجیره ای مشکل را حل نمود.

درهم سازی Hashing

درهم سازی

- ✓ تابع hash
  - تابعی است که یک کلید را به عنوان ورودی گرفته و یک آدرس را بر می گرداند.
  - یکی از توابع hash، تابع باقیمانده تقسیم است.
  - $h(k)=k \bmod m$
  - m نباید توانی از ۲ باشد. یک عدد اول که نزدیک به توانی از دو نباشد، مناسب است.
- ✓ ضریب بارگذاری
  - جدول درهم سازی T به اندازه m، n عنصر را در خود ذخیره می کند.
  - $\frac{n}{m}$ را ضریب بارگذاری (load factor) گویند و با α نشان داده می شود.
- ✓ روش زنجیره ای برای حل برخورد
  - در روش زنجیره ای (chaining) رکوردهای تصادفی به یکدیگر زنجیر می شوند.
  - [illegible] T[h(k)] [illegible] h(k) [illegible]

درهم سازی Hashing

درهم سازی

- ✓ روش آدرس دهی باز برای حل برخورد
  - در آدرس دهی باز (open hashing) از یک آرایه به اندازه m استفاده می شود که حداکثر m عنصر در آن جای می گیرد.
  - در هر درایه فقط یک عنصر قرار میگیرد.
  - باید حداکثر تعداد عناصر از قبل مشخص باشد یا از جدول درهم سازی پویا (dynamic hash table) استفاده شود.
  - جدول درهم سازی پویا جدولی است که برحسب نیاز کم یا زیاد می شود. (نصف یا دو برابر)
  - با شروع از محل collision جستجوی خطی به سمت انتهای فایل شروع شدهو رکورد تصادفی در اولین محلی که جا باشددرج می شود.
  - به صورت حلقوی عملیات انجام می شود.

درهم سازی Hashing

درهم سازی

- ✓ روش درهم سازی پویا برای حل برخورد
- در بیشتر کاربردها تعداد عناصر از قبل مشخص نمی باشند و با اعمال درج و حذف در هر لحظه تغییر می کند.
- جدول درهم سازی پویا جدولی است که برحسب نیاز کم یا زیاد می شود. (نصف یا دو برابر)
- عملیات های زیر انجام می شود:
- درج ساده: جدول جای خالی دارد و عنصر مانند قبل درج می شود.
- حذف ساده: حذفی که باعث تغییر در اندازه جدول نمی شود.
- درج با گسترش: جدول پر است. اندازه جدول دوبرابر می شود، عناصر موجود به جدول جدید منتقل می شوند و سپس عنصر مورد نظر در جدول جدید درج می شود.
- حذف و فشرده سازی: بعد از حذف، نسبت تعداد عناصر به اندازه جدول کم می شود. اندازه جدول نصف شده و همه عناصر به جدول جدید منتقل می شوند.

درهم سازی Hashing

درهم سازی

- ✓ روش درهم سازی پویا برای حل برخورد (درج)
- فرض می شود اندازه جدول همیشه توانی از ۲ است.
- ابتدا اندازه جدول صفر است، جدولی با یک درایه ایجاد شده و عنصر درج می شود.
- اگر تعداد عناصر موجود در جدول برابر با اندازه جدول باشد، درج با گسترش انجام می شود.
- جدولی جدید به اندازه دو برابر جدول موجود ایجاد می شودو همه عناصر موجود در جدول با تابع hash جدید در جدول جدید درج شده و جدول قبل آزاد می شود.
- جدول جدید تغییر نام داده و پارامترهای آن آزاد می شود.