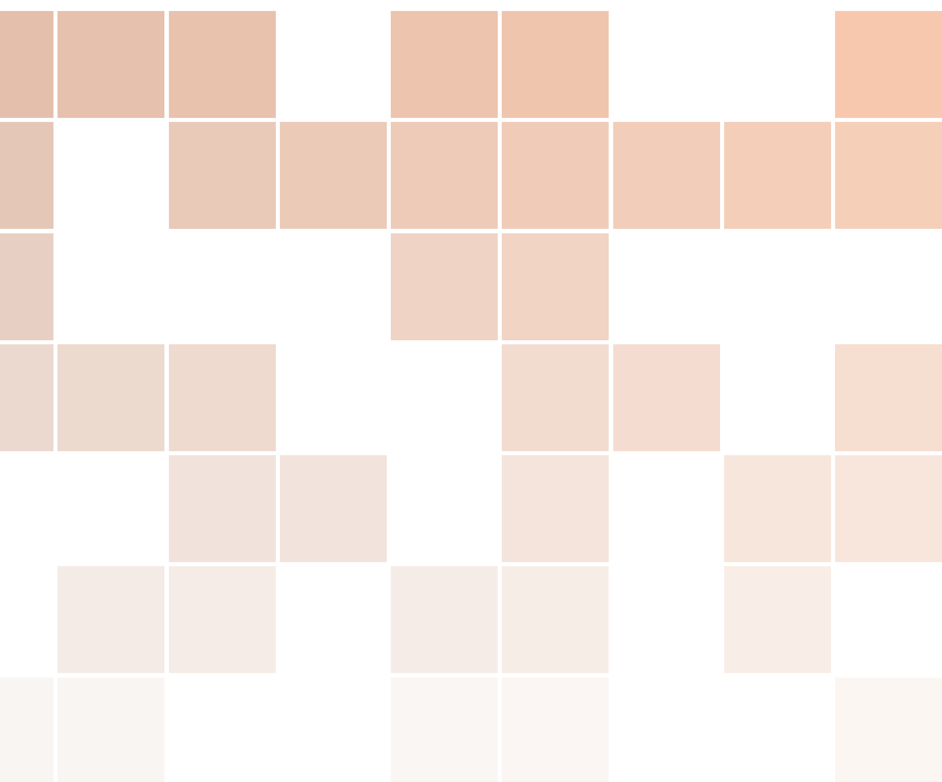
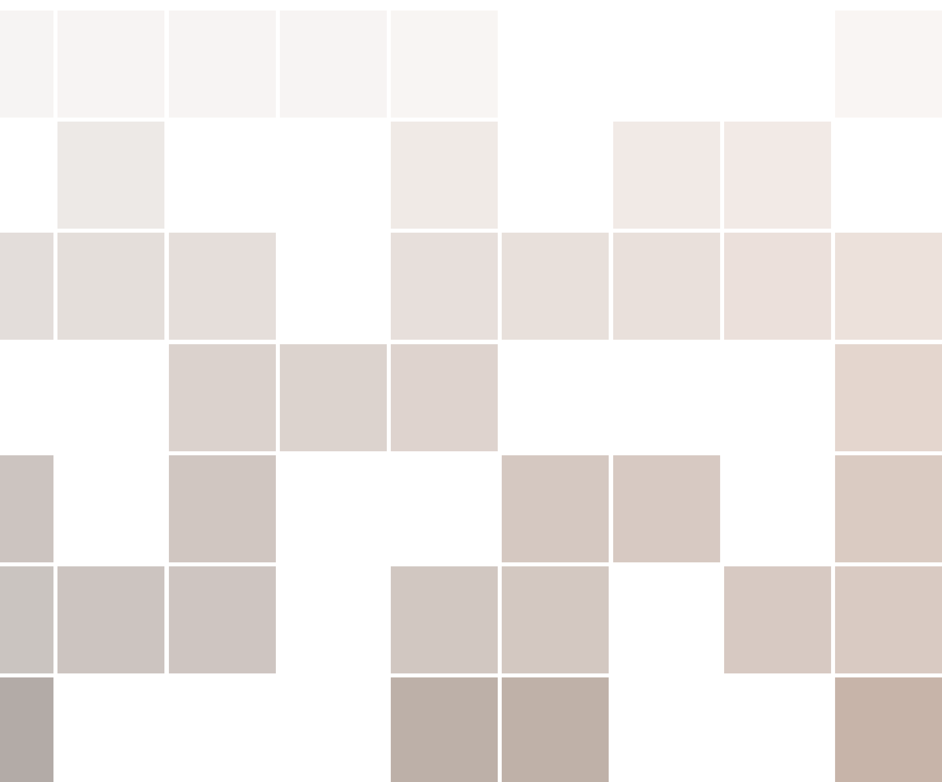


مبانی ساختمان‌های داده

رحمت قاسمی

۲۴ آبان ۱۳۹۶



فهرست مطالب

۱ مقدمه **۱**
۱.۱ انواع داده های ساده ۱
۲.۱ رکورد و یونیون ۱
۳.۱ اشاره‌گر ۳
۱.۳.۱ عملگرهای مربوط به اشاره‌گر ۳
۲.۳.۱ اشاره‌گر و تخصیص حافظه ۴
۴.۱ آرایه ۵
۱.۴.۱ پیاده‌سازی آرایه ۵
۲.۴.۱ محاسبه آدرس یک عنصر ۵
۵.۱ تمرینات فصل ۶

۲ لیست **۹**
۱.۲ مقدمه ۹
۲.۲ لیست پیوندی یک‌طرفه ۹
۱.۲.۲ مقداردهی اولیه لیست ۱۲
۲.۲.۲ درج عنصر در لیست ۱۳
۳.۲.۲ پیمایش لیست ۱۴
۴.۲.۲ حذف از لیست یک‌طرفه ۱۵
۵.۲.۲ جستجو در لیست ۱۶
۳.۲ لیست حلقوي ۱۶
۴.۲ لیست پیوندي دو طرفه ۱۸
۱.۴.۲ درج در لیست دوطرفه ۱۸
۲.۴.۲ حذف از لیست دوطرفه ۱۹
۵.۲ پیاده‌سازی لیست به کمک آرایه ۱۹
۶.۲ تمرینات فصل ۲۰

۳ پشته **۲۳**
۱.۳ عملیات تعریف شده برای پشته ۲۳
۲.۳ پیاده‌سازی پشته توسط آرایه ۲۵
۱.۲.۳ عملیات init ۲۶
۲.۲.۳ عملیات isEmpty ۲۶
۳.۲.۳ عملیات isFull ۲۶

۴.۲.۳ عملیات push ۲۶
۵.۲.۳ عملیات pop ۲۷
۳.۳ کاربرد پشته ۲۸
۱.۳.۳ فراخوانی توابع ۲۸
۲.۳.۳ ارزشیابی عبارات محاسباتی ۳۰
۳.۳.۳ تبدیل عبارت میانوندی به پسوندی ۳۲
۴.۳ تمرینات فصل ۳۶

۴ صف ۳۹
۱.۴ پیاده سازی صف خطی ۴۰
۱.۱.۴ آماده‌سازی صف ۴۰
۲.۱.۴ خالی بودن و پربودن صف ۴۱
۳.۱.۴ درج در صف ۴۱
۴.۱.۴ حذف از صف ۴۱
۲.۴ صف حلقوی ۴۲
۱.۲.۴ پیاده سازی صف حلقوی ۴۲
۳.۴ تمرینات فصل ۴۳

۵ درخت ۴۵
۱.۵ ذخیره درخت در حافظه ۴۶
۱.۱.۵ روش پرانتز ۴۶
۲.۱.۵ روش پیوندی ۴۷
۲.۵ درخت دودویی ۴۷
۱.۲.۵ عملیات قابل تعریف روی درخت دودویی ۵۰
۲.۲.۵ ذخیره درخت دودويي در حافظه ۵۰
۳.۲.۵ استفاده از آرایه برای ذخیره درخت دودویی ۵۰
۳.۵ پیمایش درختان دودويي ۵۶
۴.۵ رسم درخت با داشتن پیمایشها ۶۰
۵.۵ درخت جستجوي دودويي ۶۱
۱.۵.۵ جستجو در BST ۶۲
۲.۵.۵ درج در BST ۶۲
۳.۵.۵ حذف یک گره از BST ۶۴
۶.۵ هرم ۶۶
۱.۶.۵ ایجاد هرم بیشینه ۶۶
۲.۶.۵ درج در هرم بیشینه ۶۹
۳.۶.۵ حذف از هرم بیشینه ۷۰
۴.۶.۵ افزایش کلید ۷۱
۵.۶.۵ مرتب سازی هرمی ۷۲
۷.۵ تمرینات فصل ۷۲
۸.۵ درخت دودویی به کمک آرایه ۷۷
۹.۵ ساختار هرم دودویی ۷۹
۱۰.۵ مرتب‌سازی هرمی ۸۱

فهرست تصاویر

۱.۲ ساختار گره در لیست یک‌طرفه ۱۰
۲.۲ یک لیست پیوندی یک‌طرفه ۱۱
۳.۲ نحوه ذخیره شدن لیست در آرایه ۱۹

۱.۳ یک پشته حاوی سه عنصر، اشاره‌گر top به عنصر بالای پشته اشاره می‌کند ۲۳
۲.۳ عملیات push و pop بر روی پشته ۲۴
۳.۳ ارزیابی عبارت پسوندی + ۵ * ۳ ۴ . . . . ۳۲

۱.۴ صف حلقوی پر ۴۳

۱.۵ درخت عمومی با یک ریشه و n زیر درخت ریشه‌دار ۴۵
۲.۵ یک درخت از درجه ۴، گره f بیشترین درجه را در بین گرهها دارد . . ۴۶
۳.۵ ذخیره درخت در آرایه ۴۷
۴.۵ ساختار گره در درخت عمومی ۴۷
۵.۵ درخت فرزند چپ- همزاد راست مربوط به شکل ۲.۵ . . . . ۴۷
۶.۵ درختان دودویی نابرابر ۴۸
۷.۵ درخت دودویی پر به عمق ۳ . . . . ۴۹
۸.۵ درخت اندیس گذاری شده برای ذخیره سازی در آرایه ۵۱
۹.۵ یک درخت اریب به چپ، اندیس گذاری شده ۵۱
۱۰.۵ یک درخت کامل به عمق ۳ . . . . ۵۲
۱۱.۵ ساختار هر گره در روش پیوندی ذخیره سازی درخت دودویی ۵۴
۱۲.۵ درخت مثال ۳.۵ . . . . ۵۹
۱۳.۵ پنج حالت مختلف برای یک درخت با سه گره و پیمایش میان‌ترتیب ABC در زیر هر درخت پیمایش پس‌ترتیب آن نوشته شده است. . . . ۶۱
۱۴.۵ یک درخت جستجوی دودوئی با ۶ گره ۶۲
۱۵.۵ درج کلید ۱۲ در درخت جستجوی شکل ۱۴.۵ . . . . ۶۴
۱۶.۵ هرم بیشینه مربوط به مثال ۱.۶.۵ . . . . ۶۹
۱۷.۵ درج کلید ۷۰ در هرم شکل ۱۶.۵ قبل از اعمال جابجایی ۷۰
۱۸.۵ درج کلید ۷۰ در هرم شکل ۱۶.۵ بعد از انجام جابجایی ۷۰
۱۹.۵ حذف از هرم شکل ۱۸.۵، حذف از ریشه انجام می‌شود ۷۱

فهرست تصاویر

فهرست جداول

۱.۱ انواع داده‌ها در زبان c ۲

۱.۳ نمایش عبارات محاسباتی ۳۰
۲.۳ جدول اولویت عملگرها ۳۴

فهرست جداول

فهرست شبه‌کدها

۱.۱ تعریف ساختار person ۲
۲.۱ مقداردهی فیلدهای ساختمان ۲
۳.۱ استفاده از یونیون ۳
۴.۱ استفاده از اشاره‌گر و دسترسی غیر مستقیم به متغییر ۳
۵.۱ تخصیص حافظه ۴
۶.۱ آزاد سازی حافظه ۴
۱.۲ تعریف گره ۱۰
۲.۲ تعریف لیست ۱۰
۳.۲ ماکروهای تعریف شده برای لیست ۱۰
۴.۲ ماکروهای مربوط به ایجاد و آزاد سازی یک گره از لیست پیوندی . . ۱۱
۵.۲ عملیات ایجاد یک گره از لیست پیوندی ۱۲
۶.۲ آماده سازی لیست ۱۲
۷.۲ درج یک گره بعد از گره p ۱۳
۸.۲ عملیات درج یک گره در لیست پیوندی یک‌طرفه ۱۳
۹.۲ یافتن گره پایانی در لیست پیوندی یک‌طرفه ۱۳
۱۰.۲ درج درانتها و ابتدای لیست پیوندی یک‌طرفه ۱۴
۱۱.۲ پیمایش لیست ۱۴
۱۲.۲ استفاده از لیست در یک برنامه ۱۴
۱۳.۲ حذف اولین گره از لیست ۱۵
۱۴.۲ حذف گره بعد از یک گره از لیست پیوندی یک‌طرفه ۱۵
۱۵.۲ جستجوی یک آیتم در لیست پیوندی یک‌طرفه ۱۶
۱۶.۲ ایجاد گره در لیست حلقوی یک‌طرفه ۱۶
۱۷.۲ عملیات درج در لیست حلقوي یکطرفه ۱۷
۱۸.۲ عملیات حذف در لیست پیوندي حلقوي ۱۷
۱۹.۲ ساختار لیست پیوندی دوطرفه ۱۸
۲۰.۲ درج یک گره در لیست پیوندی دوطرفه ۱۸
۲۱.۲ درج یک گره قبل از یک گره در لیست دوپیوندی ۱۹
۲۲.۲ حذف از لیست دوپیوندی ۱۹
۲۳.۲ استفاده از آرایه برای ذخیره لیست ۱۹
۲۴.۲ تعریف لیست در ساختار آرایه‌ای ۲۰
۲۵.۲ تعریف ماکروهای مورد نیاز در لیست ذخیره شده در آرایه ۲۰

۱.۳ تعریف ساختار پشته ۲۵
۲.۳ عملیات init, isEmpty, isFull ۲۶
۳.۳ عملیات درج در پشته (push) ۲۷
۴.۳ عملیات حذف از پشته (pop) ۲۷
۵.۳ استفاده از پشته در یک برنامه نمونه ۲۷
۶.۳ تابع فاکتوریل به صورت بازگشتی ۲۹
۷.۳ پشته فراخوانی برای fact(5) ۲۹
۸.۳ تابع بازگشتی محاسبه جمله nام دنباله فیبوناتچی ۲۹
۹.۳ الگوریتم ارزیابی عبارت پسوندی ۳۱
۱.۴ ساختار صف خطی ۴۰
۲.۴ آماده‌سازی صف ۴۰
۳.۴ بررسی خالی‌بودن و پربودن صف ۴۱
۴.۴ درج در صف ۴۱
۵.۴ حذف از صف ۴۲
۶.۴ تعاریف مربوط به صف حلقوی ۴۲
۱.۵ پیاده سازی درخت به کمک آرایه ۵۲
۲.۵ ماکروهای تعریف شده برای درخت دودویی در روش ذخیره‌سازی آرایه ۵۲
۳.۵ ایجاد درخت دودویی ۵۳
۴.۵ درج در درخت دودویی ۵۳
۵.۵ ساختار درخت دودویی به روش پیوندی ۵۴
۶.۵ ماکروهای تعریف شده برای درخت دودویی به روش پیوندی ۵۴
۷.۵ ایجاد درخت دودویی به روش پیوندی ۵۵
۸.۵ درج در درخت دودویی به روش پیوندی ۵۵
۹.۵ درج گره ریشه در درخت دودویی به روش پیوندی ۵۶
۱۰.۵ ملاقات فرضی یک گره ۵۶
۱۱.۵ پیمایش میان‌ترتیب درخت دودویی ۵۷
۱۲.۵ پیمایش پیش‌ترتیب درخت دودویی ۵۷
۱۳.۵ پیمایش پس‌ترتیب درخت دودویی ۵۷
۱۴.۵ پیمایش میان‌ترتیب یک گره به صورت غیربازگشتی ۵۷
۱۵.۵ پیمایش سطحی درخت دودویی ۵۸
۱۶.۵ جستجو در درخت جستجوی دودویی ۶۲
۱۷.۵ درج در درخت جستجوی دودویی ۶۳
۱۸.۵ یافتن عنصر ماکزیمم و مینیمم ۶۴
۱۹.۵ حذف از درخت جستجوی دودویی ۶۵
۲۰.۵ عملیات $Heapify$ ۶۷
۲۱.۵ تبدیل آرایه به یک هرم ۶۷
۲۲.۵ عملیات درج در هرم ۶۹
۲۳.۵ عملیات حذف از هرم ۷۰
۲۴.۵ عملیات افزایش کلید در هرم ۷۱
۲۵.۵ مرتب سازی هرمی ۷۲

۲۶.۵ درخت دودویی به کمک آرایه ۷۷
۲۷.۵ ساختار هرم دودویی ۷۹
۲۸.۵ پیاده‌سازی مرتب سازی هرمی ۸۱

پیشگفتار

برای حل مسائل نیاز به طراحی یک الگوریتم مناسب و کارآمد می‌باشد. الگوریتم مجموعه دستورالعمل‌هایی است که مراحل مختلف انجام یک کار را بیان می‌کند. هر الگوریتم با توجه به ماهیت خود نیاز به ذخیره یکسری اطلاعات و پردازش روی آنها دارد. لذا نیازمند طراحی ساختارهایی برای الگوریتم هستیم. با توجه به الگوریتم و نیازمندیهای آن یک ساختار داده‌ای مناسب برای آن در نظر گرفته می‌شود.
کتاب حاضر به معرفی برخی از ساختارهای قابل استفاده در الگوریتمهای مختلف می‌پردازد. در تهیه این کتاب سعی شده است مطالب به طرز ساده‌ای بیان شود تا خواننده عزیز بتواند درک بهتری نسبت به مطالب داشته باشد.
از همکاران و دوستان محترم جناب آقای دکتر علی داد، جناب آقای دکتر مجید سهیلی و جناب آقای محمد صابر ایرجی که در تصحیح علمی این کتاب همیاری نموده‌اند تقدیر و تشکر می‌کنم. همچنین از دانشجویان عزیزی که در رفع اشکالات این کتاب یاری نموده‌اند نیز قدردانی می‌شود. ضمن سپاسگزاری، از شما خواننده گرامی این کتاب درخواست می‌شود تا نظرات و انتقادات سازنده خود را برای بهتر نمودن کیفیت کتاب ارسال فرمایید. لذا آدرس پست الکترونیکی اینجانب به آدرس ghasemi@iauneka.ac.ir برای این منظور ارایه می‌شود.

رحمت قاسمی
زمستان ۱۳۹۵

فصل ۱

مقدمه

ساختمان داده ها، روشی برای سازماندهی داده‌ها برای انجام هر چه بهتر عملیات پردازش روی آن می‌باشد. اگر قرار است کاری انجام شود معمولا سعی می‌شود از منابعی که در اختیار است به طور بهینه استفاده شود. چون باید برای مصرف منابع هزینه پرداخت نماییم. این منابع ممکن است زمان پردازنده، حافظه مورد نیاز برای انجام کار، میزان پهنای باند شبکه، میزان مراجعه به دیسک و یا مواردی مشابه باشد. همچنین همیشه منابع به اندازه کافی در اختیار ما وجود ندارد لذا الگوریتم بایستی با مصرف بهینه منابع بتواند جواب مناسب را با توجه به میزان منابع در اختیار، تولید نماید.

۱.۱ انواع داده های ساده

در زبانهای برنامه نویسی برای ذخیره اطلاعات در حافظه انواع داده های ساده تعریف شده است. مثلا در زبان C انواع داده ای $char$، int برای ذخیره اعدد صحیح و انواع داده ای $float$، $double$ برای ذخیره اعداد اعشاری تعریف شده است. علاوه برآن در زبان c امکان استفاده از پیشوند های $signed$، $unsigned$، $short$، $long$ نیز وجود دارد. در جدول ۱.۱ انواع داده‌ای قابل استفاده در زبان c مشخص شده است.

۲.۱ رکورد و یونیون

رکورد یا ساختمان، یک ساختار مرکب است و از تعدادی مولفه تشکیل شده است که هر مولفه آن از طریق نام در دسترس است. به هر مولفه رکورد فیلد[1] گفته می‌شود. فیلدهای رکورد از نظر نوع متفاوت می‌باشند و هر فیلد می‌تواند خود یک رکورد باشد. در زبان c از struct برای معرفی رکورد استفاده می‌شود. به طور مثال می‌توان رکورد اطلاعات فردی را به صورت آنچه که در شبه‌کد ۱.۱ نشان داده شده است، تعریف کرد. رکورد تعریف شده برای person از سه مولفه یا فیلد تشکیل شده است که عبارتند

[1] Field

جدول ۱.۱: انواع داده‌ها در زبان c

نوع داده	حدپایین	حدبالا	طول بر حسب بایت
char	۱۲۸-	۱۲۷	۱
unsigned char	۰	۲۵۵	۱
int	۳۲۷۶۸-	۳۲۷۶۷	۲
unsigned int	۰	۶۵۵۳۵	۲
long int	۲۱۴۷۴۸۳۶۴۸-	۲۱۴۷۴۸۳۶۴۷	۴
unsigned long int	۰	۴۲۹۴۹۶۷۲۹۵	۴
float	$۳/۴ * ۱۰^{-۳۸}$	$۳/۴ * ۱۰^{+۳۸}$	۴
double	$۱/۷ * ۱۰^{-۳۰۸}$	$۱/۷ * ۱۰^{+۳۰۸}$	۸
long double	$۳/۴ * ۱۰^{-۴۹۳۲}$	$۱/۱ * ۱۰^{۴۹۳۲}$	۱۰

از name, age, weight. میزان حافظه مصرفی برای متغیرهای از نوع person در حافظه برابر مجموع حافظه مصرفی هر یک از فیلدها می‌باشد، که در این مثال برابر ۲۴ می‌باشد. فیلدهای رکورد در حافظه به صورت پشت سر هم ذخیره می‌شوند. شبه‌کد ۲.۱ چگونگی مقداردهی به فیلدهای رکورد در زبان C را نشان می‌دهد.

شبه‌کد ۱.۱: تعریف ساختار person

```
1  struct person{
2          char name[20];
3          int  age;
4          int  weight;
5  };
```

شبه‌کد ۲.۱: مقداردهی فیلدهای ساختمان

```
1  struct person p1;
2  p1.age=25;
3  p1.weight=65;
4  strcpy(p1.name,"reza");
```

یونیون مقداری است که قالبهای نمایشی متفاوتی دارد. زمانی که قرار باشد به یک محل حافظه برای مقاصد مختلفی رجوع داشته باشیم از یونیون استفاده می‌کنیم. در زبان c از union برای تعریف یونیون استفاده می‌شود.
در شبه‌کد ۳.۱ نحوه تعریف یک یونیون برای ذخیره آدرس ip نشان داده شده است. این تعریف نشان می‌دهد که آدرس ip را می‌توان به صورت یک عدد ۳۲ بیتی و یا

یک آرایه ۴ عنصری از اعداد در نظر گرفت.

شبه‌کد ۳.۱: استفاده از یونیون

```
1 union {
2         long int ip;
3         char bytes[4];
4 } u;
5
6 u.bytes[0]=192;
7 u.bytes[1]=168;
8 u.bytes[2]=1;
9 u.bytes[3]=2;
```

۳.۱ اشاره‌گر

اشاره‌گر متغییری است که حاوی آدرس محلی از حافظه است. این آدرس ممکن است آدرس یک متغییر دیگر باشد. لذا اشاره‌گر روشی برای دسترسی غیرمستقیم به یک متغییر است. برای مشخص کردن یک اشاره‌گر در زبان C از علامت * در هنگام تعریف متغییر استفاده می‌شود.

مثال ۱.۱:به نحوه تعریف اشاره‌گر p در مثال زیر توجه کنید:

int x;
int * p;

در تعریف ارایه شده، x یک متغییر از نوع int است و p، یک اشاره‌گر از نوع * int است.

۱.۳.۱ عملگرهای مربوط به اشاره‌گر

دو عملگر در ارتباط با اشاره‌گر در زبان C ارایه شده است: عملگر * و عملگر &. عملگر * محتویات محلی از حافظه که اشاره‌گر بدآنجا اشاره می‌کند را برمی‌گرداند و عملگر & آدرس یک متغییر را مشخص می‌کند. شبه‌کد ۴.۱ نحوه استفاده از اشاره‌گر را نشان می‌دهد. در اینجا یک متغییر به نام x و یک اشاره‌گر به نام p تعریف شده است. در خط سوم آدرس متغییر x در اشاره‌گر p ذخیره شده است. در خط ۴ مقدار ۶ در محلی که اشاره‌گر p مشخص می‌کند ذخیره می‌شود. این محل همان آدرس متغییر x است. بنابراین دستور printf در خط ۵ مقدار ۶ را چاپ خواهد نمود.

شبه‌کد ۴.۱: استفاده از اشاره‌گر و دسترسی غیر مستقیم به متغییر

```
1 int x;
```

```
2 int *p;
3 p= &x;
4 *p=6;
5 printf("%d", x);
```

۲.۳.۱ اشاره‌گر و تخصیص حافظه

از اشاره‌گرها می‌توان در کار با حافظه پویا استفاده نمود. در برخی موارد لازم است یک برنامه در هنگام اجرای نیاز به حافظه اضافی داشته باشد. بنابراین بایستی روال تخصیص حافظه را فراخوانی نماید. در زبان c این کار به کمک رول malloc امکان پذیر است. malloc آدرس فضای مورد نیاز را به برنامه در صورت موفقیت تحویل می‌دهد و اگر این تخصیص موفق نباشد مقدار NULL را برمی‌گرداند. خروجی تابع malloc یک اشاره گر از نوع * void است. بنابراین در هنگام استفاده از این تابع بایستی عمل تبدیل نوع به نوع اشاره‌گر در حال استفاده را انجام داد. برای نمونه در قطعه کد زیر ۱۰۰ بایت حافظه درخواست و آدرس آن در اشاره‌گر p ذخیره شده است. در شبه‌کد ۵.۱ نحوه تخصیص ۱۰۰ بایت حافظه توسط تابع malloc نوشته شده است. آدرس تخصیص داده شده در متغییر اشاره‌گری p ذخیره می‌شود.

شبه‌کد ۵.۱: تخصیص حافظه

```
1 int *p;
2 p=(int *) malloc(100);
```

بعد از استفاده تخصیص حافظه پویا و استفاده از فضای حافظه در صورتی که دیگر به آن فضای حافظه نیاز نباشد لازم است فضای مربوطه را به کمک تابع free به سیستم عامل پس داد. این کار بسادگی قابل انجام است. شبه‌کد pointer نحوه استفاده از تابع free را نشان می‌دهد.

شبه‌کد ۶.۱: آزاد سازی حافظه

```
1 free(p);
```

از آنجایی که در زبان c مکانیزمی برای کنترل اشاره‌گرها ارایه نشده است، لذا برنامه‌نویس بایستی دقت کافی در استفاده از آن به عمل آورد. استفاده نادرست از اشاره‌گرها ممکن است باعث بروز اشکالاتی در برنامه شود. البته در برخی زبانهای برنامه‌سازی مکانیزمهایی برای کنترل اشاره‌گر و حافظه‌های تخصیص یافته وجود دارد.

۱.۴ آرایه

آرایه مجموعه‌ای از عناصر می‌باشد که هر عنصر از طریق اندیس عددی در دسترس می‌باشند و این دسترسی از نوع تصادفی می‌باشد، یعنی زمان دسترسی به هر یک از عناصر آرایه ثابت است. منظور از دسترسی عملیات خواندن و نوشتن در یک اندیس خاص از آرایه می‌باشد. آرایه می‌تواند یک‌بعدی باشد، که ساده‌ترین نوع آرایه است و یا چند بعدی باشد، که در نوع دوبعدی به آن ماتریس می‌گویند.

۱.۴.۱ پیاده‌سازی آرایه

ساده ترین روش پیاده سازی آرایه، ترتیبی از خانه‌های پشت سر هم حافظه میباشد، که در زبانهای مختلف برنامه نویسی وجود دارد. اندیس شروع آرایه ممکن است صفر (مثلا در زبان c) و یا یک (مثلا در زبان پاسکال) باشد. از یک طرف دیگر مایل هستیم بدانیم که یک آرایه چگونه در حافظه ذخیره می شود و علاوه بر آن آدرس یک عنصر خاص آرایه در حافظه چند است. همانطور که می دانیم حافظه اصلی کامپیوتر یک آرایه یک بعدی بسیار بزرگ است. زبانهای برنامه سازی آرایه را به دو روش در حافظه قرار می دهند، که عبارتند از:

- روش سطری
- روش ستونی

از هر روش ذخیره‌سازی که استفاده شود در نهایت آرایه به صورت یک آرایه یک‌بعدی در حافظه ذخیره خواهد شد و کامپایلر زبان برنامه‌نویسی به طور اتوماتیک محاسبه آدرس را با توجه به اندیس‌های داده شده در برنامه انجام می‌دهد.

روش سطری

در روش سطری ذخیره سازی نگاه ما ابعاد آرایه به صورت سطری است، به طور مثال اگر یک آرایه دو بعدی با m سطر و n ستون داشته باشیم در آن صورت در روش سطری ذخیره سازی ما یک آرایه یک بعدی داریم با m عنصر که هر عنصر آن یک آرایه n عنصری است. همین دیدگاه نیز برای آرایه‌های چند بعدی حاکم است.

روش ستونی

در روش سطری ذخیره سازی نگاه ما ابعاد آرایه به صورت ستونی است، به طور مثال اگر یک آرایه دو بعدی با m سطر و n ستون داشته باشیم در آن صورت در روش ستونی ذخیره سازی ما یک آرایه یک بعدی داریم با n عنصر که هر عنصر آن یک آرایه m عنصری است.

۲.۴.۱ محاسبه آدرس یک عنصر

در ادامه فرض ما برآن است که روش ذخیره سازی آرایه به صورت سطری باشد. در آن صورت معادلاتی را برای محاسبه آدرس یک عنصر آرایه ارائه می‌نماییم.

اگر آرایه یک‌بعدی باشد محاسبه آدرس کار ساده‌ای است. در یک آرایه یک بعدی، اگر α آدرس اولین عنصر در آرایه يك بعدي باشد آنگاه آدرس iامین عنصر در آرایه يك بعدي از فرمول زیرمحاسبه مي شود:

$$address(i) = \alpha + i * sizeof(array_type) \quad (۱.۱)$$

که $array_type$ نوع داده‌ای عناصر آرایه را مشخص می‌کند.

مثال ۲.۱:یک آرایه یک‌بعدی در زبان c به صورت زیر تعریف شده است.

```
int z[50];
```

اگر آدرس شروع آرایه در حافظه برابر ۲۰۰۰ باشد، آدرس عنصر ۵ این آرایه را محاسبه نمایید.

$addrress(5) = 2000 + 5 * sizeof(int) = 2000 + 5 * 2 = 2010$

در آرایه چند بعدی هر عنصر آرایه خود یک آرایه است. در این صورت تعداد عناصر آرایه از فرمول زیر محاسبه مي‌شود.

$$N = \prod_i u_i \quad (۲.۱)$$

که u_i مشخص کننده بعد iام آرایه ، $\prod$ نماد حاصلضرب است.

مثال ۳.۱:تعدادعناصر آرایه $A[۱۰][۵][۶]$ برابراست با: ۳۰۰=۶*۵*۱۰

مثال ۴.۱:فرض کنیم یک آرایه با ۵ سطر و ۴ ستون را داشته باشیم. اگرآدرس اولین عنصر آرایه برابر با ۲۱۰ باشد آنگاه آدرس عنصر در سطر ۳ و ستون ۲ رابدست آورید نوع عناصر آرایه را از نوع float فرض کنید.

حل: چون در هر سطر چهار عنصر داریم و قبل از ذخیره سازی سطر سوم دو سطر قبلی با چهار عنصر ذخیره شده است پس آدرس شروع سطر سوم آرایه در حافظه برابر است با:

$A = 210 + ((3 - 1) * 4) * sizeof(float) = 210 + 8 * 4 = 224$

حال آدرس ستون دوم از سطر سوم بر اساس معادله (۱.۱) برابر است با:

$address(2) = A + 2 * sizeof(float) = 224 + 2 * 4 = 232$

۵.۱ تمرینات فصل

تمرین ۱.۱: یک آرایه در زبان c به صورت

```
int A[10][5][4][6];
```

تعریف شده است، اگر آدرس شروع آرایه در حافظه برابر ۳۰۰ باشد، مطلوبست محاسبه آدرس A[5][2][3][4]. ■

تمرین ۲.۱: پروژه: برنامه‌ای برای محاسبه !۹۹ بنویسید. ■

تمرین ۳.۱: ساختارهای مورد نیاز برای اشکال هندسی (مانند نقطه، خط، مستطیل، دایره، مثلث) را در زبان C تعریف کنید. ■

تمرین ۴.۱: به کمک یونیون یک ساختار واحد برای مشخص نمودن دایره (مختصات مرکز و طول شعاع) و مربع (مختصات یکی از گوشه‌ها و طول ضلع) تعریف کنید. ■

فصل ۲
لیست

۱.۲ مقدمه

لیست یک مجموعه متناهی از اقلام داده ها میباشد. لیست ممکن است حاوی مقادیر تکراری باشد، ولی در مجموعه مقدار تکراری مجاز نمی‌باشد. پیاده سازی لیست می‌تواند به صورت لیست پیوندی یک‌طرفه، دوطرفه، حلقوی و یا به کمک ساختار آرایه باشد باشد. عملیات زیر روی لیست قابل تعریف است:

۱. ایجاد و آماده سازی لیست.

۲. درج یک گره جدید در لیست.

۳. حذف یک گره از لیست.

۴. جستجو در لیست.

۵. پیمایش لیست.

۶. مرتب سازی لیست.

در ادامه روشهای مختلف پیاده‌سازی لیسست مطرح شده است.

۲.۲ لیست پیوندی یک‌طرفه

لیست پیوندی یک روش پیوندی برای ذخیره لیست در حافظه می باشد، در این روش ذخیره سازی، لیست، از تعدادی گره تشکیل شده است که هر گره قلم داده‌ای و آدرس عنصر بعدی(و یا قبلی) را در خود ذخیره کرده است. در این حالت می توان دریافت که ترتیب عناصر در لیست از طریق اشاره‌گرها مشخص می‌شود و عناصر منطقا پشت سر هم، لزوماً در خانه‌هاي پشت سرهم حافظه قرار ندارند.

شکل ۱.۲: ساختار گره در لیست یک‌طرفه

هر عنصر لیست پیوندي یک‌طرفه، که به آن گره[1] مي‌گوییم، ساختاری است که در این ساختار يك فیلد اشاره‌گر به نام next وجود دارد که آدرس عنصر بعدي را در خود ذخیره مي کند، شکل ۱.۲ نمایی از این ساختار را نشان میدهد.

يك گره لیست پیوندي در زبان C به صورت زیر تعریف مي شود.

شبه‌کد ۱.۲: تعریف گره

```
1 #define INT int
2 #define pnode struct node *
3 struct node {
4         INT data;
5         pnode next;
6 };
7 typedef struct node node;
```

در این ساختار فرض شده است که data یک عنصر از نوع داده‌ای int باشد. هرچند data می‌تواند هر نوع دادهای دیگر چه ساده و چه مرکب باشد. در لیست پیوندي يك طرفه، هر گره فقط آدرس گره بعدي را ذخیره مي کند. همچنین آدرس اولین گره در اشاره‌گر start ذخیره میشود. بنابراین ساختار لیست پیوندی یک‌طرفه به صورت زیر تعریف میشود.

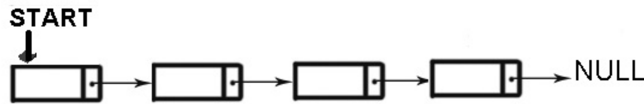
شبه‌کد ۲.۲: تعریف لیست

```
1 #define plist struct list *
2 struct list {
3         pnode start;
4 };
5 typedef struct list list;
```

به منظور خوانایی بیشتر می‌توان تعریف ماکروهای زیر را به تعریف لیست اضافه نماییم.

شبه‌کد ۳.۲: ماکروهای تعریف شده برای لیست

[1] node



شکل ۲.۲: یک لیست پیوندی یک‌طرفه

```
1 #define DATA(p)          p->data
2 #define NEXT(p)          p->next
3 #define PREV(p)          p->prev
4 #define KEY(p)           p->data
5 #define START(L)         L->start
```

اگر p یک گره از لیست باشد، هر یک از ماکروهای فوق را می‌توان به صورت زیر معرفی نمود:

- DATA(p) اطلاعات همراه گره را نشان میدهد.
- NEXT(p) آدرس گره بعدی گره p را مشخص می‌کند.
- PREV(p) آدرس عنصر قبلی گره p را مشخص می‌نماید. این ماکرو در لیست دو طرفه استفاده می‌شود.
- KEY(p) کلید ذخیره شده در گره را مشخص می‌نماید.
- START(L) آدرس اولین گره در لیست را مشخص می‌کند. که در اینجا L نماد یک لیست می‌باشد.

اگر هيچ عنصري در ليست وجود نداشته باشد يعني ليست تهي است و مقدار start برابر NULL ميباشد. ۲.۲ يك ليست پیوندی یک‌طرفه را نشان می‌دهد.
نکته مهم در استفاده از لیست پیوندی این است که برای ایجاد هر گره از لیست بایستی از حافظه پویا استفاده کنیم. پس نیاز به درخواست حافظه از سیستم عامل می باشد. براي درخواست حافظه از تابع malloc و براي پس دادن حافظه از تابع free استفاده کنیم.

شبه‌کد ۴.۲: ماکروهای مربوط به ایجاد و آزاد سازی یک گره از لیست پیوندی

```
1 #define CREATE_NEW_NODE    (node *) malloc(sizeof(node ))
2 #define FREE_NODE(p)       free(p)
```

تعریف فوق CREATE_NEW_NODE را به مجموعه تعاریف اضافه مینماید که با استفاده از تابع malloc یک محل حافظه به اندازه ساختار node اختصاص مي دهد و آدرس شروع آن را برميگرداند. چون خروجي تابع malloc از نوع * void است بنابراین

، خروجي آن را به اشاره‌گری از نوع node * تبدیل کردیم تا بتوان آدرس مورد نظر را در یک متغییر اشارهگری از نوع node * ذخیره نمود. اگر malloc نتواند حافظه مورد نیاز را تخصیص دهد، مقدار NULL را بر مي گرداند برای باز پس دادن حافظه از تابع free استفاده میشود که این کار وقتی انجام میشود که لازم باشد یک گره را از لیست حذف کنیم. عملیات ایجاد يك گره جدید با داشتن مقدار قلم داده‌ای.

شبه‌کد ۵.۲: عملیات ایجاد یک گره از لیست پیوندی

```
1  pnode create_node(INT d) {
2          pnode temp;
3          temp=  CREATE_NEW_NODE;
4          if (temp==NULL) {
5                  printf("ERROR: out of memory");
6                  exit(1);
7          }
8          DATA(temp)=d;
9          NEXT(temp)=NULL;
10         return temp;
11 }
```

در تابع create_node بایستي به مقدار مورد نیاز براي ذخیره يك گره از فضاي حافظه پویا استفاده شود. به این دلیل از ماکروی از پیش تعریف شده CREATE_NEW_NODE استفاده شده است. اگر حافظه مورد نیاز مهیا نشود در آن صورت مقدار NULL در متغییر temp قرار می‌گیرد که بایستی با یک پیام مناسب مدیریت گردد و در صورت موفق بودن عملیات درخواست حافظه محتویات گره جدید مقدار دهی میشود.

۱.۲.۲ مقداردهی اولیه لیست

مقدار دهی اولیه لیست به کمک تابع initList انجام میشود. این تابع مقادیر اولیه متغییرهای لیست را مشخص میکند. استفاده از این تابع قبل از استفاده از سایر عملیاتها ضروری میباشد.

شبه‌کد ۶.۲: آماده سازی لیست

```
1  void initList(list *L) {
2          START(L)=NULL;
3  }
```

۲.۲.۲ درج عنصر در لیست

عمليات درج يك عنصر جديد را بعد از عنصري كه آدرس آن با p مشخص شده به ليست اضافه می‌کند. این کار در دو مرحله قابل انجام خواهد بود. در مرحله اول آدرس گره بعد از p یعنی NEXT(p) را در NEXT(newnode) ذخیره میکنیم. سپس مقدار newnode را به عنوان گره بعد از p در NEXT(p) ذخیره مینماییم.

شبه‌کد ۷.۲: درج یک گره بعد از گره p

```
1 NEXT(newnode)=NEXT(p);
2 NEXT(p)= newnode;
```

پيچيدگي زماني عمليات درج درليست O(1) است.
تابع درج يك عنصر در ليست پيوندي يک‌طرفه به قرار زیر است:

شبه‌کد ۸.۲: عملیات درج یک گره در لیست پیوندی یک‌طرفه

```
1 void insert(plist L, pnode p, pnode newnode) {
2         if (p==NULL) {      // insert as first node
3                 NEXT(newnode)=START(L);
4                 START(L)=newnode;
5         } else {
6                 NEXT(newnode)= NEXT(p);
7                 NEXT(p)=newnode;
8         }
9 }
```

تابع insert یک گره جدید را که newnode نام دارد بعد از p در لیست L درج مینماید. در صورتی که مقدار p برابر NULL باشد، گره جدید به ابتدای لیست اضافه خواهد شد. خط ۲ وضعیتی است که p=NULL باشد بنابراین گره جدید در ابتدای لیست درج خواهد شد. در خطوط ۶ و ۷ عملیات درج گره جدید بعد از گره p در لیست انجام خواهد شد.

بسادگی می توان روالهای درج در انتها (append) و درج در ابتدای لیست (addFirst) را به کمک روال insert مهیا نمود که در شبه‌کد ۱۰.۲ این روالها مشخص شده است. ناگفته نماند که روال append از تابع lastNode کمک میگیرد. تابع lastNode آخرین گره لیست را مییابد.

شبه‌کد ۹.۲: یافتن گره پایانی در لیست پیوندی یک‌طرفه

```
1 pnode lastNode(plist L) {
```

```
2          pnode q=START(L);
3          while (q && NEXT(q) != NULL) q=NEXT(q);
4          return q;
5  }
```

شبه‌کد ۱۰.۲: درج درانتها و ابتدای لیست پیوندی یک‌طرفه

```
1  void  append(plist L, pnode newnode) {
2        insert( L, lastNode(L), newnode);
3  }
4
5  void addFirst(plist L,pnode newnode) {
6          insert(L,NULL,newnode);
7  }
```

۳.۲.۲ پیمایش لیست

هدف از پیمایش لیست ملاقات همه گرههای لیست به منظور انجام یک عمل خاص مثلا چاپ اطلاعات گرههای لیست است. روال printList برای این منظور تعریف شده است. تعریف این روال در شبه‌کد ۱۱.۲ آورده شده است.

شبه‌کد ۱۱.۲: پیمایش لیست

```
1  void printList(plist L) {
2          pnode p=START(L);
3          while (p) {
4                  printf("%d␣", DATA(p));
5                  p=NEXT(p);
6          }
7  }
```

قطعه برنامه کوچک درشبه‌کد ۱۲.۲ نحوه استفاده از ساختار لیست پیوندی یک‌طرفه را نشان میدهد. همانطور که در شبه‌کد ۱۲.۲ ملاحظه میشود در خط اول یک متغییر از نوع لیست به نام list1 معرفی شده است. در خط دوم به کمک تابع initList این متغییر مقداردهی اولیه شده است. در خط سوم یک گره با مقدار ۶ و در خط چهارم یک گره با مقدار ۹ به انتهای لیست اضافه شده است. در خط پنجم یک گره جدید با مقدار ۱۰ در ابتدای لیست درج شده است و در نهایت در خط ششم اطلاعات گرههای لیست توسط تابع printList چاپ میشود.

شبه‌کد ۱۲.۲: استفاده از لیست در یک برنامه

```
1  int main() {
2          list list1;
3          initList(&list1);
4          append(&list1,create_node(6));
5          append(&list1,create_node(9));
6          addFirst(&list1, create_node(10));
7          printList(&list1);
8          getchar();
9          return 0;
10 }
```

۴.۲.۲ حذف از لیست یک‌طرفه

حذف یک گره در لیست میتواند در دو حالت انجام شود. نخست حذف گره ابتدایی لیست و دوم حذف گرهای از لیست که بعد از گره مورد نظر مانند q قرار گرفته است.

گره بعداز q را در متغییر p ذخیره می کنیم و عمل حذف را روی آن انجام میدهیم. اگر p گره ابتدایی لیست باشد یعنی p=START(L) آنگاه مطابق آنچه که در خط دوم تابع deleteFirst در شبه‌کد ۱۳.۲ مشخص شده است عمل حذف انجام خواهد شد. در این صورت گره بعد از p (یعنی NEXT(p)) بعد از عملیات حذف اولین گره لیست خواهد بود.

```
1  2- START(L)=NEXT(p);
```

گام دوم: اگر p، یکی از گره ها به غیر از گره ابتدایی باشد خط دوم در تابع deleteAfter عمل حذف را بسادگی انجام میدهد. یعنی با قرار گرفتن NEXT(p) در NEXT(q) بسادگی q از لیست جدا شده و عمل free روی آن در خط سوم انجام میشود.

شبه‌کد ۱۳.۲: حذف اولین گره از لیست

```
1  void deleteFirst(plist L) {
2       pnode p=START(L);
3       START(L)=NEXT(p);
4       free(p);
5  }
```

شبه‌کد ۱۴.۲: حذف گره بعد از یک گره از لیست پیوندی یک‌طرفه

```
1 void deleteAfter(plist L, pnode q) {
2       pnode p=NEXT(q);
3       NEXT(q)=NEXT(p);
4       free(p);
5 }
```

۵.۲.۲ جستجو در لیست

تنها الگوریتم قابل استفاده در لیست پیوندی یک‌طرفه استفاده از روش جستجوی ترتیبی یا خطی است. با این اوصاف بایستی تک تک گرههای لیست از ابتدا تا انتها بررسی شود. تابع جستجوي يك عنصر در لیست پیوندي یک‌طرفه به صورت زیر پیاده سازی میشود:

شبه‌کد ۱۵.۲: جستجوی یک آیتم در لیست پیوندی یک‌طرفه

```
1 pnode searchList(plist L, INT x) {
2         pnode p=START(L);
3         while (p!=NULL) {
4                 if (KEY(p)==x) return p;
5                 p=NEXT(p);
6         }
7         return NULL;
8 }
```

۳.۲ لیست حلقوي

به منظور استفاده از مقدار NULL در اشاره‌گرهای لیست پیوندی میتوان لیست را به صورت حلقوی در نظر گرفت در این صورت آخرین گره لیست به اولین گره لیست اشاره میکند. ، اگر تابع create_node در لیست يك طرفه را باز نويسي نماييم به طوري که يك گره حلقوي به ما تحویل دهد. بنابراین دستور ;NEXT(temp)=NULL در تابع create_node را باید تغییر دهیم و قرار مي دهیم: ;NEXT(temp)=temp پس تابع create_node براي لیست حلقوي به صورت زیر پیاده سازي مي شود.

شبه‌کد ۱۶.۲: ایجاد گره در لیست حلقوی یک‌طرفه

```
1 node * create_node(INT d) {
2         node * temp;
3         temp=  CREATE_NEW_NODE;
4         if (temp==NULL) {
```

```
5                   printf("ERROR:␣out␣of␣memory");
6                   exit(1);
7           }
8           DATA(temp)=d;
9           NEXT(temp)=temp;
10          return temp;
11  }
```

شبه‌کد ۲.۱۷: عملیات درج در لیست حلقوي یکطرفه

```
1   pnode lastNode(plist L) {
2           pnode q=START(L);
3           while (q && NEXT(q) != START(L)) q=NEXT(q);
4           return q;
5   }
6
7   void insert(plist L, pnode p, pnode newnode) {
8           if (p==NULL) {
9                   if (START(L)==NULL) {
10                          START(L)=newnode;
11                  } else {
12                          NEXT(newnode)=START(L);
13                          START(L)=newnode;
14                          NEXT(lastNode(L))=newnode;
15                  }
16          } else {
17                  NEXT(newnode)= NEXT(p);
18                  NEXT(p)=newnode;
19          }
20  }
```

شبه‌کد ۲.۱۸: عملیات حذف در لیست پیوندي حلقوي

```
1   void deleteFirst(plist L) {
2        if (START(L)==NULL) return;
3        pnode p=START(L);
4        pnode q=lastNode(L);
5
6        if (p==q) START(L)=NULL;
7           else {
8                START(L)=NEXT(p);
```

```
9               NEXT(q)=START(L);
10          }
11      free(p);
12  }
13
14  void deleteAfter(plist L, pnode q) {
15      pnode p=NEXT(q);
16      if (p==START(L)) START(L)=NEXT(p);
17      NEXT(q)=NEXT(p);
18      free(p);
19  }
```

۴.۲ ليست پيوندي دو طرفه

در ليست پيوندي دو طرفه هر عنصر آدرس عنصر بعدی و آدرس عنصر قبلي را ذخيره می كند. (در صورتيكه در ليست پيوندي يك طرفه هر عنصر فقط آدرس عنصر بعدي را ذخيره می كند). در شکل زير یک لیست پیوندی دوطرفه با سه گره را مشاهده می کنید.

ساختمان داده مربوط به لیست دو طرفه:

شبه‌کد ۱۹.۲: ساختار لیست پیوندی دوطرفه

```
1  struct node {
2           INT data;
3           struct node * next ;
4           struct node * prev ;
5  };
```

۱.۴.۲ درج در ليست دوطرفه

عمليات درج يك گره جديد بعد از گره p در ليست دو پيوندی
عمليات درج در چهار مرحله انجام می شود كه به به شرح زير است.

شبه‌کد ۲۰.۲: درج یک گره در لیست پیوندی دوطرفه

```
1   PREV(t)= p;
2   NEXT(t)= NEXT(p);
3   NEXT(p) =t;
4   If (NEXT(t)) PREV(NEXT(t))=t;
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	...	999
Data			15	16	18		17			10						
Next			3	4	6		9			0						

شکل ۳.۲: نحوه ذخیره شدن لیست در آرایه

عملیات درج یک گره جدید t قبل از گره p در لیست دوپیوندی

شبه‌کد ۲۱.۲: درج یک گره قبل از یک گره در لیست دوپیوندی

```
1  NEXT(t)=p;
2  PREV(t)=PREV(p);
3  PREV(p)=t;
4  If (PREV(t)) NEXT(PREV(t))=t;
```

۲.۴.۲ حذف از لیست دوطرفه

شبه‌کد ۲۲.۲: حذف از لیست دوپیوندی

```
1  if (PREV(p)) NEXT(PREV(p))= NEXT(p);
2  if (NEXT(p)) PREV(NEXT(p))= PREV(p);
3  free(p);
```

۵.۲ پیاده‌سازی لیست به کمک آرایه

در این حالت لیست را در یک آرایه عمومی ذخیره میکنیم. رکوردهای این آرایه گرههای لیست را مشخص میکند. در شکل ۳.۲ نحوه ذخیره شدن لیست در آرایه عمومی را مشاهده می‌کنید. در اینجا ترتیب عناصر به کمک اندیس next مشخص می‌شود.
list1=start=2
items=15,16,18,17,10

شبه‌کد ۲۳.۲: استفاده از آرایه برای ذخیره لیست

```
1 #define MAX_NODES      1000
2 #define pnode int
3 struct node {
4  int data;
5  int next;
6 };
7 struct node nodes[MAX_NODES];
8 int avail=1;
```

در این ساختار فرض کرده ایم که data یک عنصر از نوع دادهای int باشد. هرچند data میتواند هر نوع دادهای دیگر چه ساده و چه مرکب باشد. در لیست پیوندي یک‌طرفه، هر گره فقط آدرس گره بعدي را ذخیره مي کند. همچنین اندیس اولین گره در متغییر start ذخیره میشود. بنابراین ساختار لیست یک‌طرفه به صورت زیر تعریف میشود.

شبه‌کد ۲۴.۲: تعریف لیست در ساختار آرایه‌ای

```
1 #define plist struct list *
2 struct list {
3         pnode start;
4 };
5 typedef struct list list;
```

به منظور خوانایی بیشتر میتوان تعاریف زیبای ماکروهای زیر را به تعریف لیست اضافه نماییم.

شبه‌کد ۲۵.۲: تعریف ماکروهای مورد نیاز در لیست ذخیره شده در آرایه

```
1 #define DATA(p)         nodes[p].data
2 #define NEXT(p)         nodes[p].next
3 #define KEY(p)          nodes[p].data
4 #define START(L)        L->start
5
6 #define CREATE_NEW_NODE   avail++
7 #define FREE_NODE(p)      ;
```

۶.۲ تمرینات فصل

تمرین ۱.۲: یک تابع به زبان بنویسید که محتویات یک لیست پیوندی یک‌طرفه را به صورت معکوس چاپ کند؟(در مورد الگوریتم خود توضیح دهید) ■

تمرین ۲.۲: تابعی بنویسید که جهت اشاره‌گرها در یک لیست پیوندی یک‌طرفه عوض کند. ■

تمرین ۳.۲: عملیات درج یک گره بعد از یک گره در یک لیست دو پیوندی را بنویسید. ■

تمرین ۴.۲: تابعی به زبان بنویسید که یک لیست یک‌طرفه که شروع آن در ذخیره شده است را به انتهای یک لیست یک‌طرفه دیگر که شروع آن با مشخص شده است اضافه کند.(۱۰)

```
1 void join( struct list *s, struct list * t)
2 {
3 ???
4 }
```

■

تمرین ۵.۲: الگوریتم الحاق دو لیست پیوندي یکطرفه را بنویسید. ■

تمرین ۶.۲: الگوریتم الحاق دو لیست پیوندی یک‌طرفه حلقوی را بنویسید. ■

تمرین ۷.۲: یک لیست یک‌طرفه موجود است، الگوریتمی برای معکوس کردن این لیست پیوندی بنویسید. ■

تمرین ۸.۲: تابعی بنویسید که با استفاده از آن بتوان تعداد گره های برگ یک درخت دودویی را مشخص نمود. زمان اجرایی این تابع چقدر است؟ ■

تمرین ۹.۲: لیست مرتب: لیستي که نودهایش ترتیب داشته باشد (چه نزولي ، چه صعودي). الگوریتم درج در این لیست را بنویسید. ابتدا لیست را پیمایش مي کنیم و عددي را که مي خواهیم به لیست اضافه میکنیم باید قبل از عددي قرار بگیرد که از آن کوچکتر باشد. ■

تمرین ۱۰.۲: دو لیست z و y مفروضند به طوري که نودهاي آنها به ترتیب صعودي مرتب است. تابعي بنویسید که دو لیست x و y را در يك لیست (z) ادغام کند به طوري که نودهاي z مرتب باشد. هیچ حافظه جدیدي درخواست نمي شود. ■

تمرین ۱۱.۲: تابعي بنویسید که گره اي که حاوي يك عدد است از يك لیست پیوندي حلقوي حذف کند.

```
1         void delete_by_key (struct list * L , int num)
```

■

فصل ۳

پشته

پشته یک لیست مرتب است که عملیات درج و حذف از یک طرف آن که تاپ(top) نامیده می شود، انجام می‌شود. آخرین عنصری که به پشته وارد شده اولین عنصری است که از پشته خارج می شود، از این جهت پشته یک ساختمان داده LIFO[۱] نامیده میشود. در شکل ۱.۳ یک پشته را مشاهده می‌کنید، که حاوی سه عنصر می‌باشد.

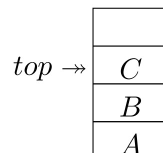
ساده‌ترین روش برای پیاده سازی پشته استفاده از آرایه است. روش دیگر استفاده از لیست پیوندی است. در حالت اول به یک آرایه برای ذخیره کردن عناصر و یک اشاره گر به عنصر بالای پشته به نام top نیاز است. وقتی که پشته خالی باشد top مقدار ۱- را نشان می دهد. در شکل ۱.۳ یک پشته را مشاهده می کنید که به ترتیب عناصر A,B,C از چپ به راست در آن درج شده اند و top به عنصر بالای پشته اشاره میکند.

۱.۳ عملیات تعریف شده برای پشته

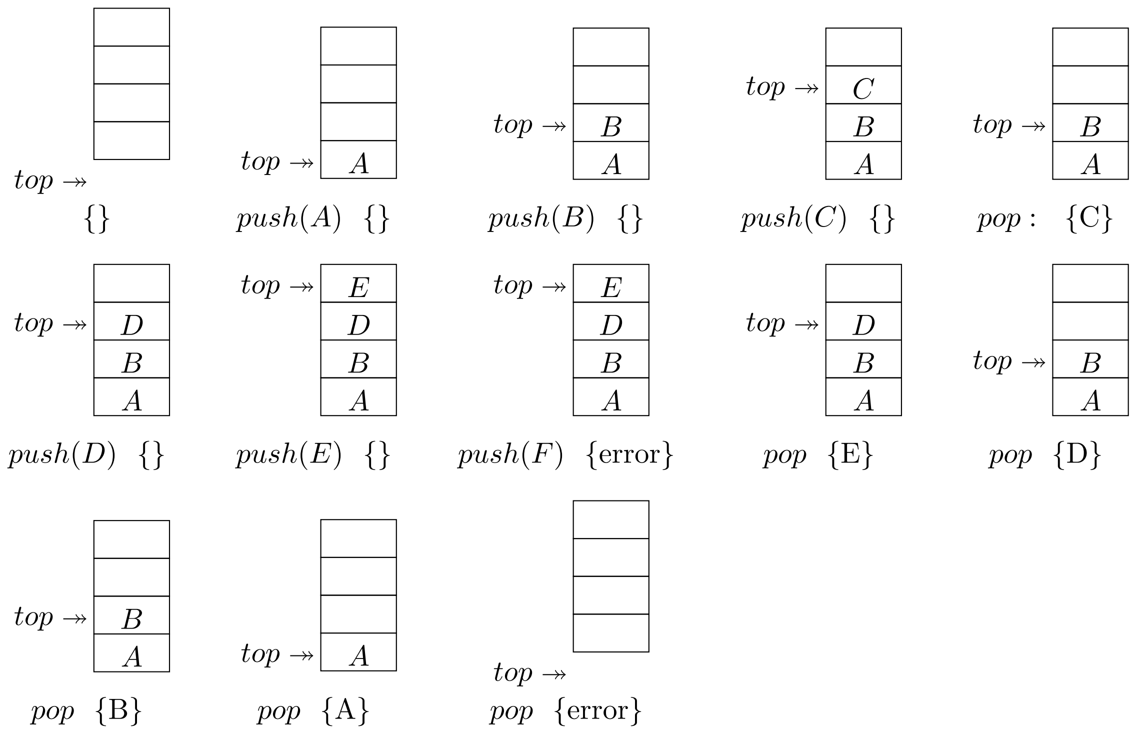
عملیاتی از قبیل init, isEmpty, isFull, push, pop برای پشته تعریف شده است، که در ادامه هر یک را شرح می‌دهیم. در هر یک از تعاریف زیر s نشان دهنده یک پشته است.

- init(s) پشته s را برای استفاده آماده می‌کند.

[۱] Last In First Out



شکل ۱.۳: یک پشته حاوی سه عنصر، اشاره‌گر top به عنصر بالای پشته اشاره می‌کند



شکل ۳.۲: عملیات push و pop بر روی پشته

- isEmpty(s) اگر پشته s خالی باشد نتیجه true در غیر اینصورت نتیجه false را نشان می‌دهد. پشته وقتی خالی است که هیچ عنصر در آن ذخیره نشده باشد.
- isFull(s) اگر پشته s پر باشد نتیجه true در غیر اینصورت نتیجه false را نشان می‌دهد. پشته وقتی پر است که دیگر نتوانیم بدلیل محدودیت حافظه عنصر جدیدی در پشته ذخیره نماییم.
- push(s,x) عنصر x را در بالای پشته درج می‌کند. و نشان‌گر top را طوری جابجا می‌کند که به عنصر درج شده جدید به عنوان عنصر بالای پشته اشاره می‌کند.
- pop(s) عنصری را از بالای پشته حذف کرده و نشان‌گر top را جابجا می‌کند، طوری که به عنصر بالای پشته جدید اشاره می‌کند.

۲.۳ پیاده‌سازی پشته توسط آرایه

در این بخش به پیاده‌سازی پشته به کمک آرایه می‌پردازیم. در این پیاده سازی فرض شده است که عناصر پشته از نوع int باشد. تعریف ساختار ذخیره سازی پشته در شبه‌کد ۱.۳ نمایان است. در خط اول تعداد عناصر پشته توسط MAX_STACK_SIZE. در خط دوم نوع عناصر پشته تعریف شده است. خطوط ۴ الی ۷ ساختار پشته را نشان می‌دهد که از یک آرایه و یک نشانگر به نام top تشکیل شده است. top اندیس عنصر بالای پشته را ذخیره می‌کند. در صورت خالی بودن پشته top مقدار ۱− را نشان می‌دهد. اشاره‌گر top از طریق ماکروی TOP در دسترس می‌باشد. و همچنین عنصر بالای پشته توسط ماکری TOP_ITEM(s) مشخص می‌شود.

شبه‌کد ۱.۳: تعریف ساختار پشته

```
1  #define MAX_STACK_SIZE 100
2  #define ITEM int
3
4  struct stack {
5     ITEM element[MAX_STACK_SIZE];
6     int top;
7  };
8  typedef struct stack stack;
9
10 #define TOP(s)          (s)->top
11 #define TOP_ITEM(s)     (s)->element[(s)->top]
```

۱.۲.۳ عملیات init

عملیات init مقادیر اولیه متغییرهای پشته s را مشخص می‌نماید و در صورت نیاز حافظه های مورد نیاز را تخصیص می‌دهد. در اینجا فقط کافی است که مقدار top برابر ۱ـ شود تا پشته آماده استفاده گردد. در شبه‌کد ۲.۳ عملیات init پیاده سازی شده است.

شبه‌کد ۲.۳: عملیات init, isEmpty, isFull

```
1  void init(stack *s) {
2          TOP(s)=-1;
3  }
4
5  int isEmpty(stack *s) {
6          return (TOP(s)==-1);
7  }
8
9  int isFull(stack *s) {
10         return TOP(s)==MAX_STACK_SIZE-1;
11 }
```

۲.۲.۳ عملیات isEmpty

این عملیات وضعیت خالی بودن پشته را نشان میدهد. اگر پشته خالی باشد نتیجه این عملیات برابر true است و اگر پشته خالی نباشد نتیجه این عملیات false است. پشته وقتی خالی است که top برابر عدد ـ ۱ باشد. در شبه‌کد ۲.۳ عملیات isEmpty پیاده‌سازی شده است.

۳.۲.۳ عملیات isFull

عملیات isFull وضعیت پر بودن پشته را مشخص می‌کند. اگر پشته پر باشد خروجی این عملیات true است و اگر پشته پر نباشد خروجی این عملیات false است. پشته وقتی پر است که به آخرین عنصر آرایه اشاره نماید. در شبه‌کد ۲.۳ عملیات isFull را مشاهده می‌نمایید.

۴.۲.۳ عملیات push

عملیات Push یک عنصر را به پشته اضافه می کند. وقتی که عملیات براي درج يك عنصر در پشته انجام مي‌شود، اگر پشته پر نباشد، ابتدا به top يك واحد اضافه مي شود وسپس آن عنصر در محلي كه top نشان مي دهد، قرار می گيرد. عملیات push را در شبه‌کد ۳.۳ پیاده‌سازی شده است. عملیات اساسی درج در خط ۶ و ۷ انجام میشود. در خط یک برای جلوگیری از عمل درج در پشته پر از تابع isFull استفاده شده است.

شبه‌کد ۳.۳: عملیات درج در پشته(push)

```
1 void push(stack *s, ITEM x){
2         if (isFull(s)) {
3                 printf("Error:␣Stack␣is␣Full.");
4                 exit(1);
5         }
6         TOP(s)++;
7         TOP_ITEM(s)=x;
8 }
```

۵.۲.۳ عملیات **pop**

عملیات pop یک عنصر را از بالای پشته حذف می کند. pop به شرطی می تواند عملیات حذف را انجام دهد که پشته خالی نباشد. بنابراین قبل از برداشتن عنصر از بالای پشته، خالی بودن پشته را به کمک تابع isEmpty بررسی می کند. اگر پشته خالی نباشد، عنصر بالای پشته در یک متغییر کمکی ذخیره می‌شود و سپس از مقدار top یک واحد کم می شود. عملیات pop در شبه‌کد ۴.۳ آورده شده است.

شبه‌کد ۴.۳: عملیات حذف از پشته(pop)

```
1 ITEM pop(stack *s) {
2         if (isEmpty(s)){
3                 printf("Error:␣Stack␣is␣Empty.");
4                 exit(1);
5         }
6         ITEM x=TOP_ITEM(s);
7         TOP(s)--;
8         return x;
9 }
```

در شبه‌کد ۵.۳ نحوه استفاده از پشته در برنامه نشان داده شده است.

شبه‌کد ۵.۳: استفاده از پشته در یک برنامه نمونه

```
1 int main() {
2         int x;
3         stack s;
4         init(&s);
5
```

```
6          push(&s, 10);
7          push(&s, 15);
8
9          x=pop(&s);
10         printf("%d", x);        // prints 15
11
12         pop(&s);
13         printf("%d", x);        // prints 10
14
15         return 0;
16 }
```

۳.۳ کاربرد پشته

پشته یکی از مهمترین ساختمان داده‌ها می‌باشد که کاربردهای فراوانی دارد. برای مثال می توان به نقش پشته به عنوان قسمتی از حافظه برنامه اشاره نمود که اگر نباشد امکان استفاده از زیر برنامه‌های بازگشتی و حتی فراخوانی توابع در برنامه‌ها وجود ندارد. از جمله کاربردهای پشته می توان به موارد زیر اشاره نمود:

- فراخوانی توابع.
- ارزشیابی عبارات محاسباتی.
- پیمایش عمقی درخت وگراف.

۱.۳.۳ فراخوانی توابع

محلی از حافظه برنامه برای پشته در نظر گرفته می‌شود که می‌تواند محل مناسبی برای ذخیره اطلاعات موقت برنامه باشد. همچنین دستورات زبان اسمبلی مانند PUSH, POP, CALL, RET و ... به استفاده از این مکان حافظه نیازمند می‌باشند. همانطور که میدانید دستور CALL در زبان اسمبلی برای فراخوانی یک زیربرنامه استفاده می‌شود. این دستور آدرس دستورالعمل بعد از خود را در پشته ذخیره می‌نماید و سپس اجرای دستورات از محل زیربرنامه آغاز میشود. در پایان زیربرنامه دستورالعمل RET قرار دارد که آدرس بازگشت را از روی پشته برمی‌دارد و اجرای برنامه از دستور بعد از دستور CALL ادامه می‌یابد. زبانهای برنامه نویسی مانند c از پشته برای ارسال پارامترهای تابع استفاده می‌کنند. برای این کار ابتدا مقادیر پارامترها در پشته قرار می‌گیرد و سپس عمل فراخوانی انجام میشود. در زبان c مقادیر پارامترها را از راست به چپ در پشته قرار می‌گیرد.

به عنوان مثال از کاربرد پشته در توابع بازگشتی به توابع بازگشتی فاکتوریل و فیبوناتچی اشاره می‌کنیم.

تابع بازگشتی فاکتوریل

در شبه‌کد ۶.۳ تابع بازگشتی فاکتوریل پیاده سازی شده است.

شبه‌کد ۶.۳: تابع فاکتوریل به صورت بازگشتی

```
1  int fact(int n) {
2    if (n<=1) return 1;
3    return n*fact( n-1);
4  }
```

شبه‌کد ۷.۳: پشته فراخوانی برای fact(5)

```
1  fact(5)= 5*fact(4)
2  fact(4)= 4*fact(3)
3  fact(3)= 3*fact(2)
4  fact(2)= 2*fact(1)
5  fact(1) = 1
6  fact(2)= 2*1=2
7  fact(3)= 3*2=6
8  fact(4)= 4*6=24
9  fact(5)= 5*24=120
```

دنباله فیبوناتچی

دنباله فیبونانچی دنباله‌ای از اعداد به صورت زیر است:

۱ ۱ ۲ ۳ ۵ ۸ ۱۳ ...

جمله صفر و یک در این دنباله برابر یک هستند، و بقیه جملات از جمع دو جمله قبلی محاسبه می‌شوند. اگر F_n نماد جمله nام دنباله فیبوتاتچی باشد در آنصورت می‌توانیم بنویسیم:

$F_0 = 1, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

در شبه‌کد ۸.۳ یک تابع بازگشتی برای محاسبه جمله nام دنباله فیبوناتچی نوشته شده است. هر چند می‌توان یک الگوریتم غیر بازگشتی نیز برای این محاسبه پیاده‌سازی کرد.

شبه‌کد ۸.۳: تابع بازگشتی محاسبه جمله nام دنباله فیبوناتچی

```
1  int fibo(int n)
2  {
3          if (n<=1) return 1;
```

جدول ۱.۳: نمایش عبارات محاسباتی

میانوندی	$a+b$
پسوندی	$a\ b\ +$
پیشوندی	$+\ a\ b$

```
4         else retuen fibo(n-1)+fibo(n-2);
5 }
```

در شبه‌کد ۸.۳ یک پیاده سازی از تابع بازگشتی محاسبه F_n را مشاهده‌می‌نمایید. نکته حائز اهمیت آن است که استفاده از پشته در فراخوانی بازگشتی تابع منجر به ساده شدن پیاده‌سازی تابع و درک ساده آن می‌شود، اما نباید فراموش کنیم که هر فراخوانی بازگشتی تابع از فضای پشته سیستم برای ذخیره پارامترهای ارسالی، مقادیر بازگشتی و آدرس بازگشت استفاده می‌کند. لذا اگر تعداد این فراخوانی‌ها افزایش پیدا کند ممکن است منجر به پر شدن پشته سیستم و به اصطلاح سرریز پشته شود. می‌توان نتیجه گرفت استفاده از فراخوانی بازگشتی نیاز به استفاده از فضای بیشتری از حافظه سیستم به دلیل استفاده از پشته سیستم است.

۲.۳.۳ ارزشیابی عبارات محاسباتی

عبارتهای محاسباتی به سه صورت قابل نمایش است:

۱. روش میانوندی[۲]: در این روش عملگر در بین عملوندها قرار می‌گیرد.

۲. روش پسوندی[۳]: در این روش عملگر بعد از عملوندها قرار می‌گیرد.

۳. روش پیشوندی[۴]: در این روش عملگر قبل از عملوندها قرار می‌گیرد.

به طور مثال عبارت $a+b$ را می توان در هر یک از سه روش فوق به صورت جدول ۱.۳ نشان داد.

در روش میانوندی برای به هم زدن اولویت عملگرها باید از پرانتز استفاده شود، ولی در روشهای پسوندی و پیشوندی نیاز به پرانتزگذاری عبارت محاسباتی نیست.

ارزیابی یک عبارت پسوندی

به عنوان یک مثال دیگر از کاربرد پشته ها می توان به استفاده از آن در ارزیابی عبارت پسوندی اشاره نمود. الگوریتم ارزیابی یک عبارت پسوندی در شبه شبه‌کد ۹.۳ نشان داده شده است. ورودی این الگوریتم یک عبارت پسوندی است. نحوه کار این الگوریتم بدین صورت است که در ابتدا یک پشته خالی در نظر گرفته می‌شود.

---

[۲] infix
[۳] postfix
[۴] prefix

شبه‌کد ۹.۳: الگوریتم ارزیابی عبارت پسوندی

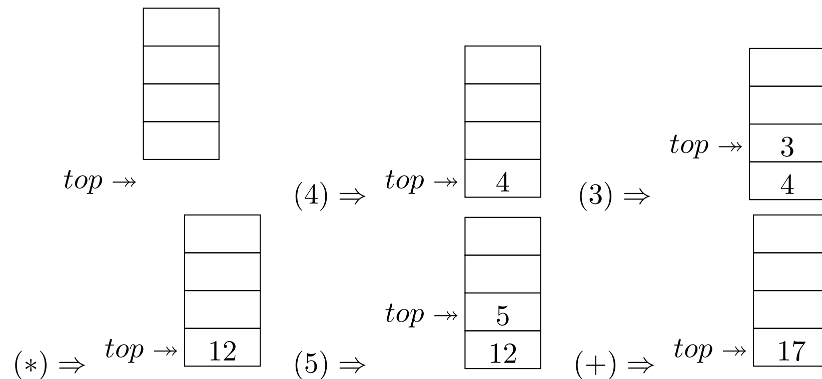
```
1  algorithm EvalPostfix( postfix ) {
2     init(stack s)
3
4     token= extract_token_from_postfix;
5     while(token <> nil) {
6          if (token is number)  push(s, token);
7          else
8          if (token is operator) {
9            x1=pop(s);
10           x2=pop(s);
11           if (token=='+') z=x2+x1;
12           if (token=='-') z=x2-x1;
13           if (token=='*') z=x2*x1;
14           if (token=='/') z=x2/x1;
15           push(s,z);
16         }
17         token= extract_token_from_postfix;
18    }
19   return pop(s);
20 }
```

سپس در هر تکرار یک نشانه از عبارت پسوندی جدا می‌شود. اگر نشانه عدد باشد، در پشته درج می‌شود. اگر نشانه یک عملگر باشد در آنصورت دو مقدار از پشته برداشته می‌شود و پس از تاثیر عملگر نتیجه مجددا در پشته درج می‌شود. تکرار تا رسیدن به پایان جمله پسوندی ادامه خواهد داشت. در پایان تکرار مقدار ارزیابی شده در بالای پشته قرار دارد.

مثال ۱.۳ :می‌خواهیم عبارت پسوندی + ۵ * ۳ ۴ را به کمک پشته ارزیابی نماییم. مراحل انجام ارزیابی در شکل ۳.۳ مشخص شده است. در ابتدا پشته خالی است. سپس ۴و ۳ چون عملوند هستند وارد پشته می‌شوند. با رسیدن به عملگر * دو عملوند از پشته خارج شده و سپس نتیجه عمل ضرب یعنی ۱۲ وارد پشته می‌شوند. با ادامه یافتن الگوریتم در نهایت مقدار ۱۷ به عنوان نتیجه عملیات در بالای پشته قرار دارد.

ارزیابی یک عبارت میانوندی

برای ارزیابی یک عبارت میانوندی که ممکن است حاوی پرانتز باشد می توانیم ابتدا عبارت پسوندی معادل آن را محاسبه کرد و سپس عملیات ارزیابی را با استفاده از الگوریتم ارزیابی عبارت پسوندی که قبلا بدان اشاره شد، انجام داد. پس مساله اصلی در اینجا تبدیل عبارت میانوندی به معادل پسوندی آن است. در بخش بعدی نحوه تبدیل یک عبارت میانوندی به عبارت پسوندی تشریح شده است.



شکل ۳.۳: ارزیابی عبارت پسوندی + ۵ * ۳ ۴

۳.۳.۳ تبدیل عبارت میانوندی به پسوندی

روش پرانتز:
مراحل تبدیل عبارت میانوندی به عبارت پسوندی به شرح زیر است:

۱. ابتدا جمله میانوندی را به طور کامل با توجه به اولویت عملگرها پرانتزگذاری می‌کنیم.

۲. هر عملگر را بعد از پرانتز بسته مربوط به خودش قرار می‌دهیم.

۳. پرانتزها را حذف می‌کنیم.

۴. چیزی که باقی مانده است عبارت پسوندی معادل می‌باشد.

مثال ۲.۳:محاسبه معادل پسوندی عبارت $a * b + c$.
ابتدا جملا را پرانتزگذاری می‌کنیم که می‌شود:

$$((a * b) + c)$$

حال عملگرها را بعد از پرانتز بسته مربوطه قرار می‌دهیم.

$$((a\ b) *\ c)+$$

پرانتزها را حذف می‌کنیم:

$$a\ b\ *\ c\ +$$

مثال ۳.۳:محاسبه معادل پسوندی عبارت $a * (b + c) - g/d$.
ابتدا جملا را پرانتزگذاری می‌کنیم که می‌شود:

$$((a*(b+c))-(g/d))$$

حال عملگرها را بعد از پرانتز بسته مربوطه قرار می‌دهیم.

$$((a\ (b\ c)+)*\ (g\ d)/)-$$

پرانتزها را حذف می‌کنیم:

$$a\ b\ c\ +\ *\ g\ d\ /\ -$$

ساده‌سازی عبارت

در این روش با تعریف متغیرهای ساده جدید جمله ورودی را آنقدر ساده می‌کنیم تا به یک عبارت ساده میانوندی برسیم. متغییرها طوری تعریف می‌شوند که لطمه‌ای به اولویت عملگره وارد نشود. بعد از اینکه به یک جمله ساده رسیدیم عمل تبدیل را انجام می‌دهیم و در هر مرحله متغییرها را به صورت پسوندی در جمله وارد می‌کنیم.

مثال ۴.۳:محاسبه معادل پسوندی عبارت $a*b+c$.

$$i=a*b+c$$
$$X=a*b\Rightarrow i=X+c$$
$$i=X\ c+=a\ b\ *\ c+$$

مثال ۵.۳:محاسبه معادل پسوندی عبارت $a*(b+c)-g/d$.

$$i=a*(b+c)-g/d$$
$$X=b+c, Y=g/d\Rightarrow i=a*X-Y$$
$$Z=a*X\Rightarrow i=Z-Y$$
$$i=Z\ Y\ -$$
$$=a\ X\ *\ Y\ -$$
$$=a\ b\ c\ +\ *\ g\ d\ /\ -$$

استفاده از پشته

در این روش از یک پشته برای ذخیره کردن عملگرهای ورودی استفاده می شود. در این الگوریتم باید اولویت عملگرها را نیز در نظر بگیریم. اولویت جمع و تفرق برابر است. اولویت ضرب و تقسیم هم برابر است. اولویت ضرب و تقسیم از جمع و تفریق بیشتر است. پرانتز باز هم به عنوان یک شبه عملگر در نظر گرفته میشود. اولویت پرانتزباز بیرون پشته از همه عملگرها بیشتر است، ولی اولویت پرانتز باز درون پشته از همه عملگرها کمتر است. اینگونه دریافت میشود که پرانتز باز داری اولویت دوگانه است که ممکن است کمی گیج کننده باشد و مشکلاتی را در پیاده سازی الگوریتم به وجود آورد. پیشنهاد ما استفاده از علامت دیگری به جای پرانتز باز داخل پشته (مثلا کاراکتر '[') است. در جدول ۲.۳ اولویت عملگرها را مشخص شده است.

جدول ۲.۳: جدول اولویت عملگرها

اولویت	عملگر
۰	[
۱	$+ \; -$
۲	$* \; /$
۳	(

الگوریتم: تبدیل عبارت میانوندی به معادل پسوندی.
ورودی: عبارت میانوندی.
خروجی: عبارت پسوندی.

۱. شروع

۲. یک پشته خالی آماده کن.

۳. عبارت میانوندی از چپ به راست مورد بررسی قرار می گیرد.

۴. یک نشانه از عبارت میانوندی جدا کن.

۵. اگر نشانه، عملوند باشد آن را در خروجی بنویس.

۶. اگر نشانه عملگر باشد آنگاه:

(آ) اگر اولویت نشانه از اولویت عملگر بالای پشته بیشتر بود آنگاه نشانه را به پشته اضافه کن(پرانتزباز به صورت '[' وارد پشته می‌شود) و برو به مرحله ۸.

(ب) یک عملگر از بالای پشته خارج کن و درخروجی جمله پسوندی بنویس و برو به مرحله (آ).

۷. اگر نشانه پرانتز بسته بود آنگاه تا رسیدن به یک علامت پرانتز باز عملگر از پشته خارج و در خروجی جمله پسوندی بنویس. پرانتز باز، را هم از پشته خارج کن.

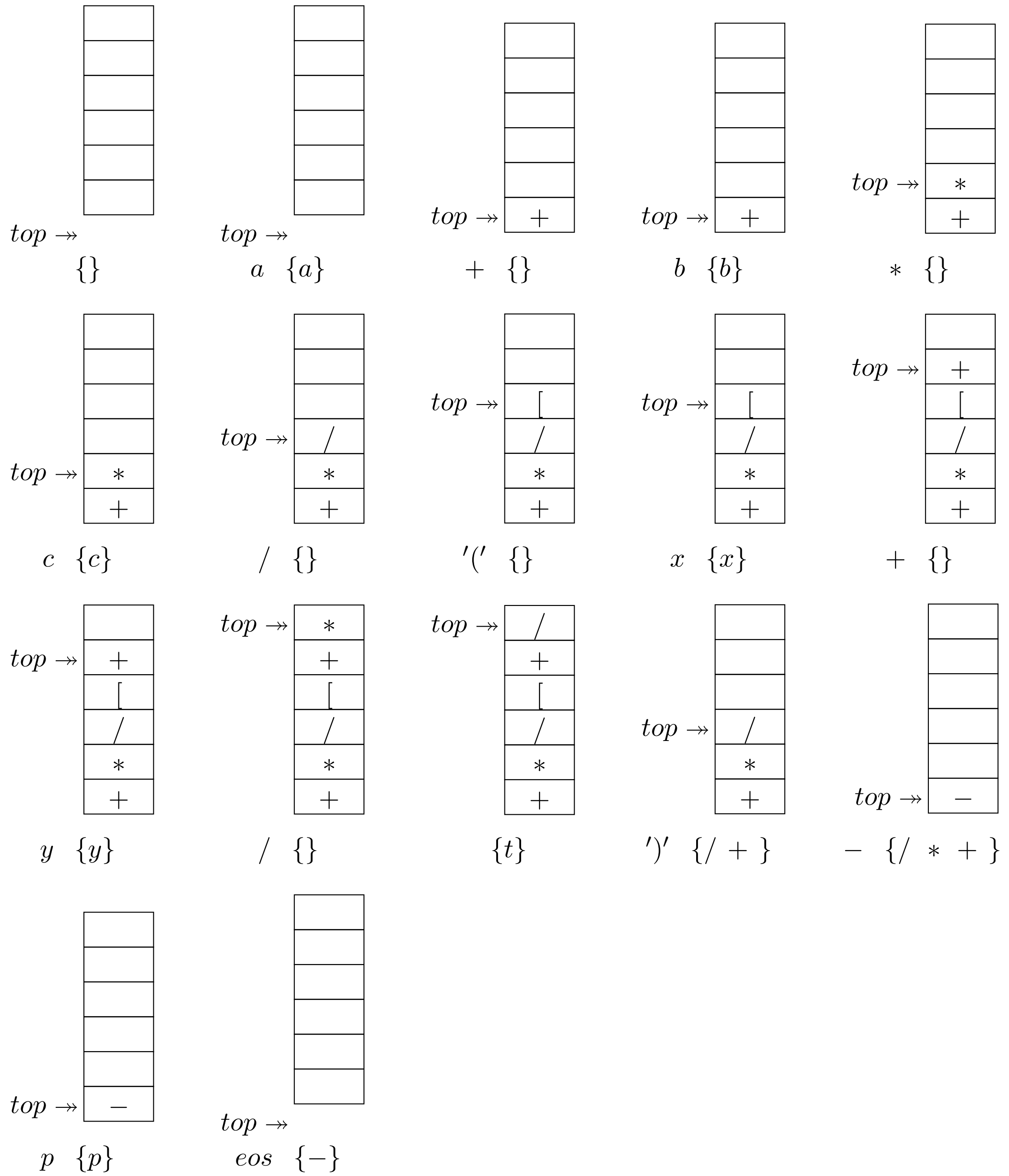
۸. اگر به انتهای جمله نرسیدی برو به مرحله ۴.

۹. همه عملگرها را از پشته خارج و در خروجی جمله پسوندی بنویس.

۱۰. پایان

مثال ۶.۳:می‌خواهیم معادل پسوندی عبارت $a + b * c/(x + y/t) - p$ را به کمک پشته بدست آوریم. در هر مرحله تکرار در شکل زیر ورودی و خروجی در پایین هر

شکل مشخص شده است، خروجی در درون علامت نوشته شده است. نتیجه هر تکرار در پشته اعمال شده است.



حال خروجی هر تکرار را اگر به ترتیب از چپ به راست در کنار هم قرار دهیم به جمله پسوندی زیر می‌رسیم.

$$a\ b\ c\ x\ y\ t\ /\ +\ /\ *\ +\ p\ -$$

۴.۳ تمرینات فصل

تمرین ۱.۳: پشته پیوندی را به طور کامل پیاده سازی نمایید. ■

تمرین ۲.۳: دلیل خالی ماندن یک عنصر از آرایه در یک صف حلقوی چیست. ■

تمرین ۳.۳: عبارت میانوندی زیر را به مدل پسوندی و پیشوندی تبدیل کنید.

a+b/d+(x/y*k)*(c*(z + m))

■

تمرین ۴.۳: تابع **isFull** (پر بودن صف) را برای یک صف حلقوی را بنویسید؟ ■

تمرین ۵.۳: تابع **push** برای اضافه کردن یک عنصر به پشته پیوندی را بنویسید.

■

تمرین ۶.۳: نشانه گذاری پسوندی را برای عبارت زیر بنویسید؟ وضعیت پشته را در هر مرحله نشان دهید.

$(a + b) * d + e/(f + a * d) + c$

■

تمرین ۷.۳: اگر s یک پشته باشد، خروجی قطعه برنامه زیر را مشخص نمایید.

```
1  a=2; b=5;
2  push(s,a)
3  push(s,b+10)
4  push(s,3)
5  push(s,a-b)
6  push (s,a+b)
7  while (top<>-1)
8  {
9          item=pop(s);
10         printf("%d",item);
11 }
```

■

تمرین ۸.۳: عبارت زیر را با استفاده از پشته ارزشیابي کنید. وضعیت پشته را در در هر مرحله نشان دهید.

5 6 ∗ 4 + 3 9 + ∗

■

تمرین ۹.۳: يك پشته را مي توان با استفاده از يك ليست پيوندە يکطرفه پياده سازي نمود.در این پشته اشاره گري است که آدرس گره بالاي پشته را نشان مي دهد. توابع و در این پشته را پیاده سازي نمائيد. ■

تمرین ۱۰.۳: تابعي به زبان C بنويسيد كه درستي پرانتزهاي يك عبارت را بررسي كند. (با استفاده از پشته) به طور مثال عبارت $a+(C+D+(x*y))$ یک عبارت درست است و عبارت $(x+(y+z)$ یک عبارت نادرست است.

از چپ به راست عبارت را پردازش مي کنيم. اگر به پرانتز باز برسيم پرانتز را وارد پشته مي کنيم. و اگر به پرانتز برسيم يك پرانتز باز از پشته خارج ميکنيم. (اگر پشته خالي نباشد پس تاکنون جمله درست است و اگر پشته خالي باشد و نتوان از آن عنصري خارج كرد ، عبارت نادرست پرانتز گذاري شده است.) و در آخركار پشته بايد خالي باشد. خالي بودن پشته نشان مي دهد که تعداد پرانتز باز و بسته برابر است. ■

تمرین ۱۱.۳: تابعي بنويسيد كه در یک عبارت رشته‌ای، مشخص کند آیا تعداد Aها با تعداد Bها برابر است؟ (با استفاده از پشته) ■

فصل ۴

صف

صف، یک لیست مرتب است که عمل درج از یک سمت که ته‌صف [1] نامیده می‌شود و عمل حذف از سمت دیگر آن که سرصف [2] نامیده می‌شود، انجام می‌شود. صف یک ساختمان داده **FIFO**[3] است، یعنی اولین عنصری که وارد می‌شود، اولین عنصری است که از صف خارج می‌شود. اگر q نماد یک صف باشد، عملیاتی که روی صف می‌توان انجام داد عبارتند از:

- **initq(q):** صف q را آماده‌سازی می‌نماد.
- **addq(q,x):** قلم داده x را در صف q درج می‌نماید.
- **deleteq(q):** یک قلم داده از صف q حذف می‌نماید.
- **isemptyq(q):** وضعیت خالی بودن صف را نشان بررسی می‌کند. صف وقتی خالی است که هیچ قلم داده‌ای در آن نباشد.
- **isfullq(q):** وضعیت پر بودن صف را نشان بررسی می‌کند. صف وقتی پر است که فضای لازم برای درج عنصر جدید در آن نباشد.

کاربردهای متعددی برای صف وجود دارد که میتوان به موارد زیر اشاره نمود:

- پیمایش سطحی گراف.
- پردازش فرآیندهای سیستم عامل.
- هر سیستمی که نیازمند ایجاد سیستم نوبتی باشد از صف استفاده میکند.

صف در انواع مختلفی می‌تواند باشد که عبارتند از:صف خطی، صف حلقوی، صف اولویت.

---

[1] rear
[2] front
[3] First In First Out

۱.۴ پیاده سازی صف خطی

برای پیاده‌سازی صف خطی هم می‌توان از آرایه استفاده نمود و هم از لیست در اینجا از یک آرایه برای پیاده‌سازی صف استفاده می‌شود. لذا برای ذخیره عناصر در صف یک آرایه تعریف می‌شود. همچنین دو اشاره‌گر هم برای نشان دادن سرصف و ته‌صف تعریف می‌شود. این تعارف در شبه‌کد ۱.۴ مشخص شده است. در این تعریف MAX_QUEUE_SIZE حداکثر ظرفیت صف را مشخص می‌کند.

شبه‌کد ۱.۴: ساختار صف خطی

```
1  #define MAX_QUEUE_SIZE 100
2  #define ITEM int
3  struct queue{
4          ITEM    elements[MAX_QUEUE_SIZE];
5          int     front;
6          int     rear;
7  };
8  struct queue queue;
9
10 #define FRONT(q)    (q)->front
11 #define REAR(q)     (q)->rear
12 #define FRONT_ITEM(q)   (q)->elements[(q)->front]
13 #define REAR_ITEM(q)    (q)->elements[(q)->rear]
14 #define MOVE_FRONT(q)   FRONT(q)++
15 #define MOVE_REAR(q)    REAR(q)++
```

دربخش‌های بعدی عملیات تعریف شده روی صف پیاده‌سازی می‌شود.

۱.۱.۴ آماده‌سازی صف

برای اینکه صف قابل استفاده باشد، بایستی عملیات آماده سازی روی آن انجام شود. مثلا مقداردهی اولیه متغییرهای ته‌صف و سرصف و در صورت نیاز درخواست حافظه برای ذخیره عناصر، از آنجاییکه در این پیاده سازی از یک آرایه ساده استفاده نموده‌ایم پس فضای حافظه به صورت ثابت به اندازه MAX_QUEUE_SIZE در حافظه برای ما فراهم است. در شبه‌کد ۲.۴ مقداردهی اولیه به متغیرهای سرصف و ته‌صف انجام می‌شود. ورودی این تابع آدرس یک صف می‌باشد که از طریق اشاره‌گر q مشخص شده است.

شبه‌کد ۲.۴: آماده‌سازی صف

```
1  void initq(queue *q) {
```

```
2           FRONT(q)=REAR(q)=0;
3   }
```

۲.۱.۴ خالی بودن و پربودن صف

وضعیت خالی‌بودن و پربودن صف توسط ماکروهای isempty و isfull بررسی می‌شود که در شبه‌کد ۳.۴ لیست شده است. صف وقتی خالی است که ته‌صف و سرصف یک مقدار را نشان دهند. و صف وقتی پر است که ته‌صف مقدار MAX_QUEU_SIZE-1 شود.

شبه‌کد ۳.۴: بررسی خالی‌بودن و پربودن صف

```
1   #define isemptyq(q)    (FRONT(q)==REAR(q))
2   #define isfullq(q)     (REAR(q)== MAX_QUEUE_SIZE)
```

۳.۱.۴ درج در صف

عملیلت درج در صف از سمتی که اشاره‌گر ته‌صف نشان می‌دهد انجام می‌شود. درج وقتی می‌تواند انجام شود که صف پر نباشد. این عملیات در شبه‌کد ۴.۴ لیست شده است. اگر صف پر نباشد آیتم جدید در ته‌صف قرار می‌گیرد و سپس ته‌صف جابجا می‌شود.

شبه‌کد ۴.۴: درج در صف

```
1   void addq(queue *q, ITEM x) {
2           if (isfullq(q)){
3                   printf("error:␣queue␣is␣full");
4                   exit(1);
5           }
6           REAR_ITEM(q)=x;
7           MOVE_REAR(q);
8   }
```

۴.۱.۴ حذف از صف

عملیات حذف از صف در شبه‌کد ۵.۴ نشان داده شده است. حذف وقتی قابل انجام است که صف خالی نباشد. اگر صف خالی نبود ابتدا عنصر سرصف در متغییر موقتی x ذخیره می‌شود. سپس نشانگر سرصف جابجا می‌شود. در نهایت مقدار موقتی به عنوان خروجی مشخص می‌شود.

شبه‌کد ۵.۴: حذف از صف

```
1 ITEM deleteq(queue *q) {
2         if (isemptyq(q)){
3                 printf("error:␣queue␣is␣empty");
4                 exit(1);
5         }
6         ITEM x=FRONT_ITEM(q);
7         MOVE_FRONT(q);
8         return x;
9 }
```

۲.۴ صف حلقوی

مشکل صف خطی این است که با اضاف شدن عناصر به آن زمانی می رسد که صف پر میشود بنابراین دیگر نمی‌توان عنصر جدیدی را در آن درج کرد. و با خارج شدن عناصر از صف زمانی می رسد که هیچ عنصری در صف نیست یعنی سرصف با ته‌صف برابر می شود در این زمان دیگر صف قابل استفاده نخواهد بود. برای حل این مشکل صف حلقوی ارائه می شود.

۱.۲.۴ پیاده سازی صف حلقوی

با کمی تغییر در ساختار صف خطی بدون اینکه نیاز باشد توابع درج و حذف مربوط به صف خطی را دستکاری نماییم، صف حلقوی قابل پیاده‌سازی است. در شبه‌کد ۶.۴ برخی از ماکروها که در صف خطی تعریف شده بودند بازنویسی شده‌اند، بقیه ماکروها نیاز به تغییر ندارند.

شبه‌کد ۶.۴: تعاریف مربوط به صف حلقوی

```
1 #define MOVE_FRONT(q)  FRONT(q)=(FRONT(q)+1) % MAX_QUEUE_SIZE
2 #define MOVE_REAR(q)   REAR(q)=(REAR(q)+1) % MAX_QUEUE_SIZE
3 #define isemptyq(q)     (FRONT(q)==REAR(q))
4 #define isfullq(q)      ((REAR(q)+1) % MAX_QUEUE_SIZE == FRONT(q))
```

وضعیت خالی بودن صف حلقوی دقیقا مشابه صف خطی است. وضعیت پر بودن صف حلقوی کمی متفاوت است. صف حلقوی زمانی پر است که ته‌صف دقیقا قبل از سرصف باشد. در شکل ۱.۴ دو وضعیت ممکن از پربودن یک صف حلقوی نشان داده شده است.

		rear	front					
14	13		10	15	441	55	69	90

front								rear
10	12	14	189	52	26	144	22	

شکل ۱.۴: صف حلقوی پر

۳.۴ تمرینات فصل

تمرین ۱.۴: چرا در یک صف حلقوی پر که با آرایه پیاده‌سازی شده‌است، همیشه یک محل از آرایه خالی است؟ ■

تمرین ۲.۴: صف را به کمک لیست پیوندی پیاده‌سازی نمایید. ■

فصل ۵

درخت

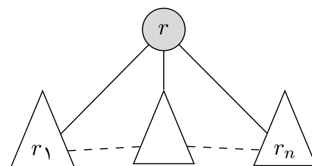
درخت مجموعه ای از یک یا چند گره می باشد که خصوصیات زیر را دارد:

- دارای یک گره به نام گره ریشه[۱] است.
- اگر $T_1..T_n$ خود درخت با ریشه $r_1..r_n$ باشند آنگاه می‌توان درختی با ریشه r ساخت که $r_1..r_n$ فرزندان r باشند. در این‌صورت $T_1..T_n$ زیر درخت‌های گره r می‌باشند.

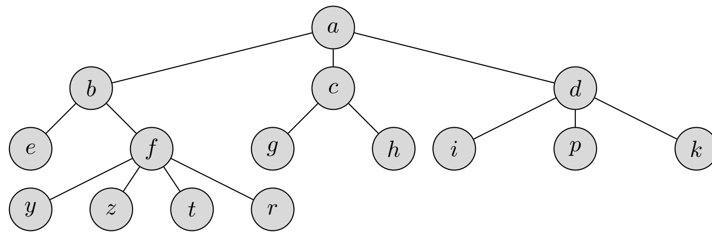
درخت یک گراف[۲] خاص است که از تعدادی گره[۳] و تعدادی یال[۴] تشکیل شده است و مسیر دوار[۵] در آن وجود ندارد، و همچنین درخت یک گراف متصل است. یال در درخت ارتباط دو گره را مشخص می‌کند، این ارتباط از نوع والد[۶]-فرزندی[۷] است. به تعداد فرزندان یک گره، درجه گره گفته می‌شود.

درجه یک درخت، بزرگ‌ترین درجه موجود در بین گره‌های درخت، است. درخت شکل ۲.۵ از درجه ۴ است. برگ یک گره فاقد فرزند از درجه صفر است. به فرزندان

[۱] root
[۲] Graph
[۳] Node
[۴] Edge
[۵] Cycle
[۶] Parent
[۷] Child



شکل ۱.۵: درخت عمومی با یک ریشه و n زیر درخت ریشه‌دار



شکل ۲.۵: یک درخت از درجه ۴، گره f بیشترین درجه را در بین گرهها دارد

یک گره گرههای همزاد[۸] گفته می‌شود. در شکل ۲.۵ گرههای b, c, d همزاد هستند، چون فرزندان گره a هستند. اجداد یک گره، گرههایی هستند که در مسیر طی شده از ریشه تا گره مورد نظر وجود دارند. مثلا a, b, e اجداد گره e هستند. عمق یا سطح یک گره، فاصله یک گره تا گره ریشه است (ریشه در سطح صفر قرار دارد). سطح گره a، صفر و سطح گرههای b, c, d برابر یک است. ارتفاع یک گره مانند p، فاصله گره p تا پایینترین برگ، یا به عبارتی دیگر بیشترین سطح در زیر درخت به ریشه p است. ارتفاع ریشه همان ارتفاع درخت است. در شکل ۲.۵، ارتفاع درخت ۳ است. مسیر، دنباله‌ای از یالهای متوالی است و شاخه، مسیری است که به یک برگ ختم می‌شود. درخت متوازن درختی است که اختلاف سطح برگهای آن حداکثر یک باشد و درخت کاملا متوازن، درختی است که اختلاف سطح برگ ها در آن صفر باشد.

۱.۵ ذخیره درخت در حافظه

روشهای مختلفی برای ذخیره درخت قابل استفاده است، که در ادامه به معرفی هر یک از روشها می‌پردازیم.

۱.۱.۵ روش پرانتز

در روش پرانتز به همراه هر گره فرزندانش را درون پرانتز قرار میدهیم. این روش برای وقتی مناسب است که مثلا بخواهیم درخت را در یک فایل ذخیره نماییم. درخت شکل ۲.۵ را می‌توان مانند رشته پرانتزی زیر نوشت:

$$a(b(e, f(y, z, t, r)), c(g, h), d(i, p, k))$$

در ادامه خواهید دید، که عبارت فوق معادل با پیمایش پیش‌ترتیب درخت است. میتوان عبارت فوق را در یک آرایه ذخیره کرد به طوری که هر گره اندیس گره والد خود را ذخیره می‌کند. این نحوه ذخیره سازی را در شکل ۳.۵ مشاهده می‌کنید.

---
[۸] sibling

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
node	a	b	e	f	y	z	t	r	c	g	h	d	i	p	k
parent	0	1	2	2	4	4	4	4	1	9	9	1	12	12	12

شکل ۳.۵: ذخیره درخت در آرایه

Data		
right	*right*	*parent*

شکل ۴.۵: ساختار گره در درخت عمومی

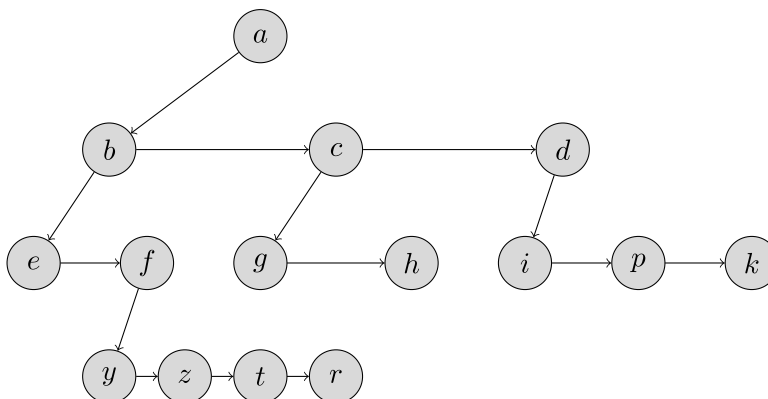
۲.۱.۵ روش پیوندی

در روش پیوندی ساختار هر گره از چهار بخش تشکیل شده است، که در یک بخش اطلاعات گره قرار دارد و سه بخش اشاره‌گری شامل یک اشاره‌گر به فرزند چپ و یک اشاره‌گر به همزاد راست و یک اشاره‌گر به والد می‌باشد. این ساختار در شکل ۴.۵ نشان داده شده است.

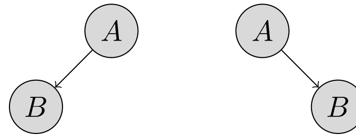
ساختار گره در روش پیوندی با ارائه ساختار پیوندی، درخت شکل ۲.۵ به صورت زیر قابل نمایش خواهد بود، که درخت فرزند چپ-همزاد راست نام دارد.

۲.۵ درخت دودویی

درخت دودویی درختی است که یا تهی است یا حاوی یک سری گرهها. هر گره حداکثر دو فرزند دارد، که یکی فرزند چپ و دیگری فرزندراست نام دارد. ترتیب گرهها در درخت دودویی مهم است. به طور مثال دو درخت زیر با هم برابر نیستند.



شکل ۵.۵: درخت فرزند چپ- همزاد راست مربوط به شکل ۲.۵



شکل ۶.۵: درختان دودویی نابرابر

قضیه ۱.۵: حداکثر تعداد گره ها در سطح iام یک درخت دودویی برابر ۲^i است. $i \geq ۰$

اثبات. این قضیه به کمک استقرا قابل اثبات است. این کار در سه گام قابل انجام است.

- (گام آغازین) اگر $i = ۰$ باشد واضح است که در سطح صفر درخت حداکثر یک گره بیشتر وجود ندارد، و آن گره ریشه است، پس حداکثر تعداد گرهها در سطح صفر برابر $۲^۰$ است.
- (فرض) فرض میکنیم در سطح $i - ۱$، حداکثر تعداد گرههای درخت برابر $۲^{i-۱}$ باشد.
- (حکم) باید ثابت کنیم حداکثر تعداد گرهها در سطح iام یک درخت دودویی برابر ۲^i است. می‌دانیم هر گره حداکثر دو فرزند دارد، پس در سطح بعدی درخت حداکثر تعداد گرهها دو برابر سطح قبلی است. بنابراین اگر n_i حداکثر تعداد گره ها در سطح iام باشد خواهیم داشت:

$$n_i \leq ۲n_{i-۱} \tag{۱.۵}$$

با توجه به فرض مرحله ۲ خواهیم داشت:

$$ni \leq ۲(۲^{i-۱}) = ۲^i \tag{۲.۵}$$

از رابطه ۲.۵ حکم ثابت می‌شود.

□

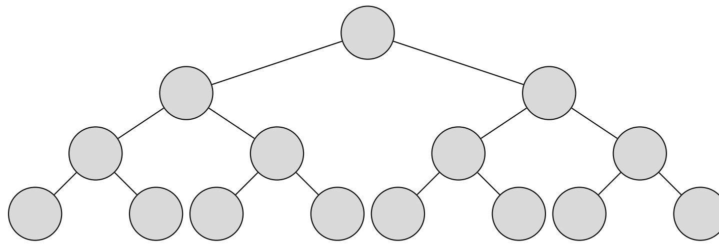
قضیه ۲.۵: حداکثر تعداد گره‌ها در یك درخت دودويي به عمق k برابر است با $۲^{k+۱} - ۱$ و $k \geq ۰$ است.

اثبات. اگر n حداکثر تعداد گره‌ها و n_i، حداکثر تعداد گره‌ها در سطح iام درخت دودویی باشند، خواهیم داشت:

$$n = n_۰ + n_۱ + \ldots + n_k = ۲^۰ + ۲^۱ + \ldots + ۲^k \tag{۳.۵}$$

با ضرب رابطه ۳.۵ در عدد دو خواهیم داشت:

$$۲n = ۲^۱ + \ldots + ۲^{k+۱} \tag{۴.۵}$$



شکل ۷.۵: درخت دودویی پر به عمق ۳

حال اگر رابطه ۳.۵ را از رابطه ۴.۵ کسر نماییم داریم:

$$۲n - n = ۲^{k+۱} - ۲^{۰}$$

پس

$$n = ۲^{k+۱} - ۱ \quad (۵.۵)$$

□

تعریف درخت پر: یک درخت دودویی به عمق k، یک درخت پر است اگر دقیقا داری $۲^{k+۱} - ۱$ گره باشد. مسلما درخت پر داری حداکثر تعداد گره‌ها در تمام سطوح خود خواهد بود و بنابراین گره از درجه یک در آن وجود ندارد و تمام گره‌های برگ در آخرین سطح درخت قرار دارند. در شکل ۱۰.۵ یک درخت پر به عمق ۳ به تصویر کشیده شده است.

قضیه ۳.۵: اگر $n_۰$ تعداد گره‌ها از درجه صفر، $n_۱$ تعداد گره‌ها از درجه ۱و $n_۲$ تعداد گره‌ها از درجه ۲ باشد آنگاه:

$$n_۰ = n_۲ + ۱ \quad (۶.۵)$$

اثبات. اگر n تعداد گره ها باشد واضح است که

$$n = n_۰ + n_۱ + n_۲ \quad (۷.۵)$$

تعداد یالهای درخت که در اینجا ما آن را با E نشان داده‌ایم را می توان از طریق رابطه ۸.۵ محاسبه کرد:

$$E = ۲n_۲ + ۱n_۱ + ۰n_۰ = ۲n_۲ + n_۱ \quad (۸.۵)$$

از طرفي هر گره به يك يال متصل است. به جز گره ریشه که به هیچ یالي متصل نیست، پس:

$$n = E + ۱ = ۲n_۲ + n_۱ + ۱ \quad (۹.۵)$$

از رابطه ۷.۵ و ۹.۵ می‌توان نتیجه گرفت:

$$n_۰ + n_۱ + n_۲ = ۲n_۲ + n_۰ + ۱ \Rightarrow n_۰ = n_۲ + ۱$$

□

۱.۲.۵ عملیات قابل تعریف روی درخت دودویی

اگر T نماد یک درخت دودویی و n یک گره از این درخت باشد، در آنصورت عملیات زیر را می‌توان برای درخت دودویی قابل تعریف است:

- **BTCreate:** یک درخت دودویی تهی ایجاد میکند.
- **ROOT(T):** ریشه درخت T را نشان می‌دهد.
- **LEFT(i):** فرزند چپ گره i را برمی‌گرداند.
- **RIGHT(i):** فرزند راست گره i را برمی‌گرداند.
- **PARENT(i):** گره والد i برمی‌گرداند.
- **DATA(T,i):** اطلاعات همراه گره i را نشان میدهد.
- **KEY(T,i):** عنصر کلیدی گره i را نشان میدهد. عنصر کلیدي در عملیات جستجو قابل استفاده است.
- **ISEMPTY(i):** تهی بودن گره i را نشان می‌دهید.
- **BTInsert(T,x,p,f):** یگ گره با محتویات x ایجاد میکند و آن را به عنوان فرزند گره p قرار می‌دهد. مقدار f نشان می‌دهد که گره جدید فرزند چپ باشد یا فرزند راست. اگر $BTcreatenode$ قادر نباشد گره جدید را ایجاد کند، (مثلا به دلیل کمبود حافظه) مقدار $NULL$ را برمی‌گرداند، در غیر اینصورت آدرس گره را برمی‌گرداند. مقدار f میتواند برابر $LEFTCHILD = ۰$ یا $RIGHTCHILD = ۱$ باشد.

۲.۲.۵ ذخیره درخت دودویي در حافظه

برای ذخیره درخت دودویی در حافظه دو روش اساسی وجود دارد که عبارتند از:

- استفاده از آرایه.
- استفاده از مدل پیوندي.

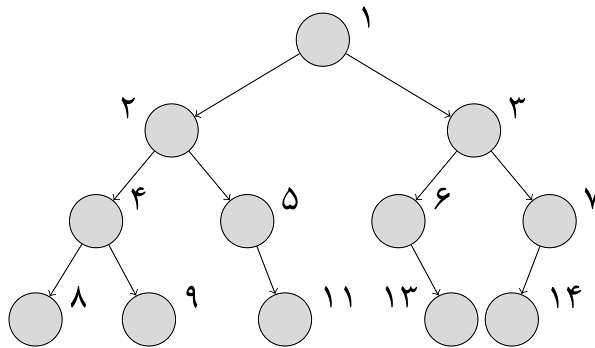
۳.۲.۵ استفاده از آرایه برای ذخیره درخت دودویی

در این روش از یک آرایه برای ذخیره اطلاعات گره‌های درخت استفاده می‌شود. قوانین زیر در ذخیره گره‌های درخت در آرایه برقرار است.

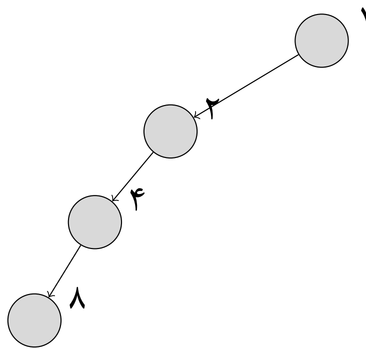
۱. ریشه در اندیس شماره یک آرایه ذخیره می‌شود.

۲. اگر i اندیس یک گره در آرایه باشد، $۲i$ اندیس فرزند چپ است و $(i > ۰)$.

۳. اگر i اندیس یک گره در آرایه باشد، $۲i+۱$ اندیس فرزند راست است و $(i > ۰)$.



شکل ۸.۵: درخت اندیس گذاری شده برای ذخیره سازی در آرایه



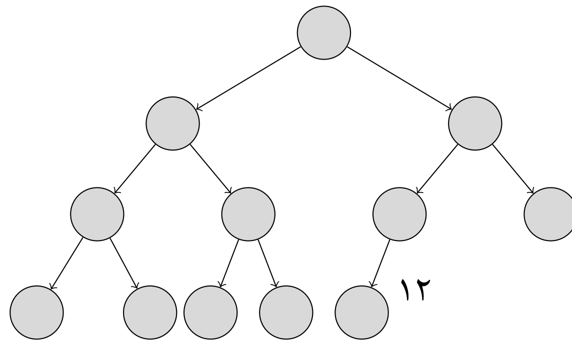
شکل ۹.۵: یک درخت اریب به چپ، اندیس گذاری شده

۴. اگر i اندیس یک گره در آرایه باشد، آنگاه $\lfloor \frac{i}{۲} \rfloor$ اندیس گره والد آن است و $(i > ۰)$.

در شکل ۸.۵ یک درخت دودویی به عمق ۳ برای ذخیره سازی در آرایه به تصویر کشیده شده است. اندیس هر گره در کنار آن نوشته شده است. همانطور که مشاهده میکنید، اندیسهای ۱۰،۱۲ و ۱۵ در آرایه استفاده نشده است.

استفاده از آرایه براي درختان پر، روش بسیار مناسبي است. ولی براي درختان اریب آرایه روش مناسبی نیست، چون بدلیل استفاده نشدن از بسیاری از اندیسهای آرایه میزان اتلاف حافظه زیاد است. در شکل ۹.۵ این موضوع به تصویر کشیده شده است.

تعریف درخت کامل: اگر تعداد گرههای درخت برابر اندیس سمت راست ترین گره در آخرین سطح درخت (که آن را آخرین گره درخت فرض میکنیم)، در ذخیره سازی به روش آرایه باشد، آنگاه با یک درخت کامل روبرو هستیم. این نشان دهنده آن است که تمام اندیسهای آرایه از اندیس اول تا اندیس آخرین گره استفاده شده است. در شکل ۱۰.۵ یک درخت کامل به عمق ۳ را می‌توانید مشاهده نمایید.



شکل ۵.۱۰: یک درخت کامل به عمق ۳

پیاده سازي روش آرایه

براي پیاده سازي درخت دودیي در آرایه تعاریف زیر را در زبان C انجام مي‌دهیم. فرض مي‌کنیم که داده‌هاي درخت از نوع int باشد و حداکثر تعداد عناصر درخت برابر ۲۵۵ باشد. با توجه به اینکه آرایه برای ذخیره اطلاعات گره‌های درخت استفاده شده است، ساختار درخت به صورت زیر تعریف میشود.

شبه‌کد ۵.۱: پیاده سازی درخت به کمک آرایه

```
1  #define         ITEM            int
2  #define         keyelement      int
3  #define         MAX             256
4  #define         NODE            int
5
6  struct tree{
7          ITEM    nodes[MAX];
8          int     root;
9          int     count;
10 };
11 typedef struct tree tree;
```

با استفاده از ماکروهاي زیباي که در شبه‌کد ۵.۲ لیست شده است، برخی از اعمال مختلف روی درخت دودویی را پیاده سازی می‌کنیم.

شبه‌کد ۵.۲: ماکروهای تعریف شده برای درخت دودویی در روش ذخیره‌سازی آرایه

```
1  #define ROOT(T)         (T)->root
2  #define LEFT(i)         (2*i)
3  #define RIGHT(i)        (2*i+1)
4  #define PARENT(i)       (i/2)
```

```
 5
 6 #define DATA(T,i)        (T)->nodes[i]
 7 #define KEY(T,i)         (T)->nodes[i]
 8 #define COUNT(T)         (T)->count
 9 #define ISVALID(i)       (i>0 && i<MAX)
10 #define ISNULL(T,i)      (!ISVALID(i) || (T)->nodes[i]== 0)
11
12 #define LEFTCHILD        0
13 #define RIGHTCHILD       1
14 #define NIL              0
```

ماكروي $ISVALID$ براي بررسي اعتبار انديس داده شده كاربرد دارد. فرض ميكنيم كه فضاي آرايه قبلا با مقدار صفر مقداردهي شده است و عدد صفر نشان دهنده خالي بودن مكان مورد نظر در آرايه مي‌باشد. اعمال $BTCreate$، $BTInsert$ را به صورت یک روال به زبان C پیاده سازی می‌کنیم. که در کدهای ۳.۵و ۴.۵ مشاهده می‌کنید.

شبه‌کد ۳.۵: ایجاد درخت دودویی

```
 1 tree * BTCreate() {
 2  tree *t;
 3  t=(tree *)malloc(sizeof(tree));
 4  if(t!=NULL){
 5          int i;
 6          for(i=0; i<MAX; i++) t->nodes[i]=0;
 7          ROOT(t)=1;
 8          COUNT(t)=0;
 9  }
10  return t;
11 }
```

شبه‌کد ۴.۵: درج در درخت دودویی

```
 1 NODE BTInsert(tree *t, dataelement x, NODE p, int f) {
 2         NODE temp= 2*p+f;
 3         if (!ISNULL(t,temp)) return 0;
 4         if (ISVALID(temp)) {
 5                 DATA(t,temp)=x;
 6                 COUNT(T)++;
 7                 return temp;
 8         } else return 0;
 9 }
```

Data		
right	left	parent

شکل ۱۱.۵: ساختار هر گره در روش پیوندی ذخیره سازی درخت دودویی

استفاده از مدل پیوندی برای ذخیره درخت دودویی

در روش پیوندی ساختار هر گره از چهار بخش تشکیل شده است، که در یک بخش اطلاعات گره قرار دارد و سه بخش اشارهگری شامل یک اشارهگر به فرزند چپ و یک اشارهگر به فرزند راست و اشاره گره به گره والد میباشد. واضح است که گره ریشه فاقد گره والد است.

شبه‌کد ۵.۵: ساختار درخت دودویی به روش پیوندی

```
1  #define  dataelement    int
2  #define  keyelement     int
3
4  typedef  struct BTnode * NODE;
5
6  struct BTnode {
7          dataelement data;
8          NODE    left;
9          NODE    right;
10         NODE parent;
11 };
12
13 struct tree {
14         NODE    root;
15         int     count;
16 };
17 typedef struct tree tree;
```

شبه‌کد ۶.۵: ماکروهای تعریف شده برای درخت دودویی به روش پیوندی

```
1  #define ROOT(T)         ((T)->root)
2  #define LEFT(i)         ((i)->left)
3  #define RIGHT(i)        ((i)->right)
4  #define PARENT(i)       ((i)->parent)
5
6  #define COUNT(T)        (T)->count
```

```
7  #define DATA(T,i)        ((i)->data)
8  #define KEY(T,i)         ((i)->data)
9
10 #define ISVALID(i)       (i!=NULL)
11 #define ISNULL(T,i)      (i==NULL)
12
13 #define LEFTCHILD        0
14 #define RIGHTCHILD       1
15 #define ROOTNODE         2
```

شبه‌کد ۷.۵: ایجاد درخت دودویی به روش پیوندی

```
1  tree *BTCreate() {
2   tree *t=(tree *) malloc(sizeof(tree));
3   if(t!=NULL) {
4           ROOT(t)=NULL;
5           COUNT(t)=0;
6   }
7   return t;
8  }
```

شبه‌کد ۸.۵: درج در درخت دودویی به روش پیوندی

```
1  NODE BTInsert(tree *t,dataelement x, NODE p, int f) {
2          NODE temp;
3          temp=(NODE )malloc(sizeof(struct BTnode));
4
5          if (ISVALID(temp)) {
6                  LEFT(temp)=NULL;
7                  RIGHT(temp)=NULL;
8
9                  if (f==LEFTCHILD) LEFT(p)= temp;
10                 else
11                 if (f==RIGHTCHILD) RIGHT(p)=temp;
12
13                 PARENT(temp)=p;
14                 DATA(t,temp)=x;
15                 COUNT(t)++;
16                 return temp;
17         }
18         return NULL;
19 }
```

شبه‌کد ۹.۵: درج گره ریشه در درخت دودویی به روش پیوندی

```
1 NODE rootInsert(tree *t, dataelement x) {
2         ROOT(t)=BTInsert(t, x, NULL, ROOTNODE);
3 }
```

۳.۵ پیمایش درختان دودويي

هدف از پیمایش ملاقات گره‌های درخت است حداکثر برای یک بار. پیمایش‌های مختلفی که روی درخت دودویی می‌توان انجام داد عبارتند از:

- پیمایش میان ترتیب[۹].
- پیمایش پیش ترتیب[۱۰].
- پیمایش پس ترتیب[۱۱].
- پیمایش سطحي[۱۲].

سه پیمایش اول را میتوان در مقایسه با گراف نوعي پیمایش عمقي درنظر گرفت و پیمایش سطحي مانند پیمایش سطحي در گراف است. پیمایشهاي اول تا سوم ابتدا فرزند چپ و سپس فرزند راست را پیمایش میکنند، اگر این ترتیب به گونه دیگري باشد سه پیمایش دیگر نیز میتوان به لیست فوق اضافه نمود. اگر منظور از L ملاقات زیر درخت چپ باشد و R ملاقات زیر درخت سمت راست و v ملاقات ریشه باشد، پیمایش میان‌ترتیب، پیش‌ترتیب و پس‌ترتیب را به ترتیب میتوان به صورت LvR، vLR و LRv در نظر گرفت. در ادامه هر یک از پیمایشهای گفته شده به زبان C پیاده سازی شده است.

شبه‌کد ۱۰.۵: ملاقات فرضی یک گره

```
1 void BTvisit(tree *t, NODE i ) {
2    printf("%d␣", DATA(t,i));
3 }
```

---

[۹] inorder
[۱۰] preorder
[۱۱] postorder
[۱۲] levelorder

شبه‌کد ۱۱.۵: پیمایش میان‌ترتیب درخت دودویی

```
1 void BTinorder (tree *t, NODE i) {
2   if (ISNULL(t,i)) return;
3   BTinorder (t,LEFT(i));
4   BTvisit(t,i);
5   BTinorder (t,RIGHT(i));
6 }
```

شبه‌کد ۱۲.۵: پیمایش پیش‌ترتیب درخت دودویی

```
1 void BTpreorder (tree *t, NODE i) {
2   if (ISNULL(t,i)) return;
3   BTvisit(t,i);
4   BTpreorder (t,LEFT(i));
5   BTpreorder (t,RIGHT(i));
6 }
```

شبه‌کد ۱۳.۵: پیمایش پس‌ترتیب درخت دودویی

```
1 void BTpostorder (tree *t, NODE i) {
2   if (ISNULL(t,i)) return;
3   BTpostorder (t,LEFT(i));
4   BTpostorder (t,RIGHT(i));
5   BTvisit(t,i);
6 }
```

همانطورکه مشاهده میشود توابع مربوط به پیمایش های عمقی درخت به صورت بازگشتی نوشته شده است ولی این امکان وجود دارد که ما این توابع را به صورت غیر بازگشتی هم بنویسیم، برای این منظور به پشته نیاز می‌باشد. الگوریتم غیربازگشتی پیمایش $inorder$ را در ۱۴.۵ نوشته شده است.

شبه‌کد ۱۴.۵: پیمایش میان‌ترتیب یک گره به صورت غیربازگشتی

```
1 void BTinorder2(tree* t,NODE i) {
2         stack s;
3         init(&s);
4         NODE x=i;
5         while( !ISNULL(t,x) || !isEmpty(&s) ) {
```

```
6           while(!ISNULL(t,x)) {
7                  push(&s, x);
8                  x=LEFT(x);
9           }
10          x=pop(&s);
11          BTvisit(x);
12          x=RIGHT(x);
13         };
14  }
```

همانطور که در ۱۴.۵ مشاهده می‌شود در ابتدا یک پشته در نظر گرفته می‌شود سپس در خط چهارم گره آغازین که ریشه درخت است در متغییر کمکی x ذخیره می‌شود. در هر مرحله از تکرار فرزندان چپ گره x به ترتیب تا سمت چپ ترین گره x، در پشته قرار می‌گیرند. در خط ۱۰، یک گره از بالای پشته خارج شده و ملاقات می‌شود، حال باید درخت سمت راست گره مورد نظر را پیمایش نماییم. که در خط ۱۲ مقدار جدید x، برابر گره سمت راست آخرین گره ملاقات شده قرار می‌گیرد. تکرار عملیات تا زمانی انجام میشود که پشته خالی نباشد یا درخت سمت راست برای آخرین گره ملاقات شده موجود باشد.

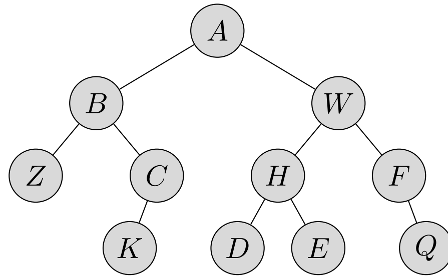
پیمایش سطحی درخت، گره های درخت را به صورت سطح به سطح ملاقات می‌کند. در پیمایش سطحی به یک صف نیازمندیم. الگوریتم مربوط به پیمایش سطحی در ۱۵.۵ آورده شده است.

شبه‌کد ۱۵.۵: پیمایش سطحی درخت دودویی

```
1   void levelorder (tree *t) {
2          NODE temp;
3          queue q;
4          initq(&q);
5          addq(&q, ROOT(t));
6          while (!isemptyq(&q)) {
7                 temp = deleteq(&q);
8                 if (!ISEMPTY(LEFT(temp)) addq(&q, LEFT(temp));
9                 if (!ISEMPTY(RIGHT(temp)) addq(&q, RIGHT(temp));
10                BTvisit(temp);
11         }
12  }
```

مثال ۱.۵: درخت شکل ۱۲.۵ را در نظر بگیرید، می‌خواهیم تمام پیمایش‌های درخت را بدست آوریم.

۱. پیمایش میان‌ترتیب درخت: پیمایش میان‌ترتیب درخت از گره ریشه آغاز می‌گردد. در هر مرحله قانون پیمایش به صورت LvR روی هر گره پیمایش نشده



شکل ۱۲.۵: درخت مثال ۳.۵

اعمال می‌شود. پایان کار وقتی است که هیچ گرهی برای ساده شدن وجود نداشته باشد.

$$\begin{aligned} inorder &= \textcircled{A} \\ &= \textcircled{B}_A\textcircled{W} \\ &= \textcircled{Z}_B\textcircled{C}_A\textcircled{H}_W\textcircled{F} \\ &= ZB\textcircled{K}CA\textcircled{D}H\textcircled{E}WF\textcircled{Q} \\ &= ZBKCADHEWFQ \end{aligned}$$

۲. پیمایش پیش‌ترتیب درخت عبارتند از:

$$\begin{aligned} preorder &= \textcircled{A} \\ &= A\textcircled{B}\textcircled{W} \\ &= AB\textcircled{Z}\textcircled{C}W\textcircled{H}\textcircled{F} \\ &= ABZC\textcircled{K}WH\textcircled{D}\textcircled{E}F\textcircled{Q} \\ &= ABZCKWHDEFQ \end{aligned}$$

۳. پیمایش پس‌ترتیب درخت عبارتند از:

$$\begin{aligned} postorder &= \textcircled{A} \\ &= \textcircled{B}\textcircled{W}A \\ &= \textcircled{Z}\textcircled{C}B\textcircled{H}\textcircled{F}WA \\ &= Z\textcircled{K}CB\textcircled{D}\textcircled{E}H\textcircled{Q}FWA \\ &= ZKCBDEHQFWA \end{aligned}$$

۴. پیمایش سطحی درخت: در پیمایش سطحی با گره ریشه شروع می‌کنیم، و در هر باز ساده‌سازی فرزندان گره را به انتهای لیست اضافه می‌کنیم. کار تا جایی ادامه پیدا

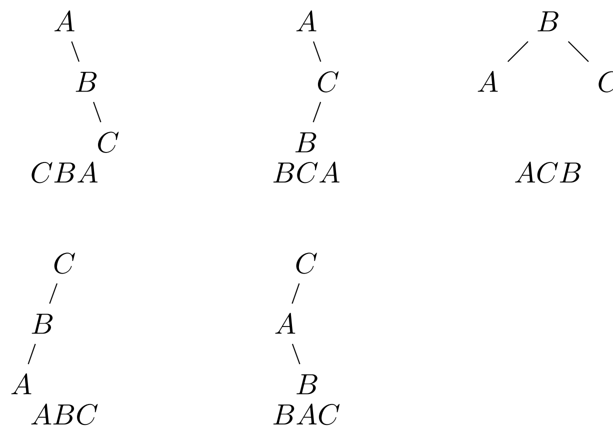
می‌کند که گرهی برای ساده شدن وجود نداشته باشد.

$$
\begin{aligned}
levelorder &= (A) \\
&= A(B)(W) \\
&= AB(W)(Z)(C) \\
&= ABW(Z)(C)(H)(F) \\
&= ABWZC(H)(F)(K) \\
&= ABWZCH(F)(K)(D)(E) \\
&= ABWZCHF(K)(D)(E)(Q) \\
&= ABWZCHFKDEQ
\end{aligned}
$$

۴.۵ رسم درخت با داشتن پیمایشها

آیا با داشتن مثلا یکی از پیمایش‌های درخت میتوان درخت را رسم کرد؟ فرض می‌کنیم پیمایش میان‌ترتیب یک درخت دودویی با ۳ گره برابر ABC باشد و بخواهیم درخت را رسم کنیم در شکل ۱۳.۵، ۵ حالت مختلف نشان داده شده است. آنچه دریافت میشود این است که با داشتن یک پیمایش تنها نمی‌توان درخت منحصر به فردی را رسم نمود.
حال به پیمایش میان‌ترتیب پیمایش پس‌ترتیب را اضافه می‌کنیم. از آنجاییکه برای هر یک از درختان پیمایش پس‌ترتیب منحصر به فردی وجود دارد، شکل ۱۳.۵ نشان میدهد، که با داشتن پیمایشهای میان‌ترتیب و پس‌ترتیب میتوان درخت منحصر به فردی را رسم کرد. روال کار بدین صورت است که:
۱. با داشتن پیمایش پس‌ترتیب ریشه درخت مشخص میشود.
۲. با داشتن پیمایش میان‌ترتیب زیردرخت سمت چپ و راست درخت مشخص میشود.
۳. مراحل ۱و ۲ را برای تمام زیر درختان موجود تکرار می‌کنیم.

در حالت کلی اگر پیمایش میانترتیب یک درخت صورت LvR و پیمایش پسترتیب همان درخت به صورت LRv باشد، که مطابق با یکی از تنها ۵ حالت مشخص شده در شکل ۱۳.۵ می‌باشد. بنابراین پیمایشهای میان‌ترتیب و پس‌ترتیب درخت منحصر به فردی را تولید میکند. یعنی در مرحله اول درخت منحصر به فردی رسم میشود. در مراحل بعدی نیز برای زیر درختان چپ و راست هم درخت منحصر به فردی رسم میشود، پس در نهایت کل درخت منحصر به فرد است. اگر $T(n)$ تعداد کل درختان دودویی با پیمایش میانترتیب یکسان با n گره باشد در آنصورت خواهیم داشت: $T(۰), T(۱) = ۱$ اگر $n > ۱$ باشد، میتوان هر بار یکی از گرهها را به عنوان ریشه درخت فرض کرد. اگر



شکل ۱۳.۵: پنج حالت مختلف برای یک درخت با سه گره و پیمایش میان‌ترتیب ABC در زیر هر درخت پیمایش پس‌ترتیب آن نوشته شده است.

شماره این گره i باشد پس $i-۱$ گره در زیر درخت سمت چپ و $n-i$ گره در زیر درخت سمت راست وجود دارند. بنابراین:

$$T(n)=\sum_{i=۱}^{n}T(i-۱)T(n-i) \qquad (۱۰.۵)$$

حل رابطه ۱۰.۵ به رابطه ساده زیر منجر خواهد شد:

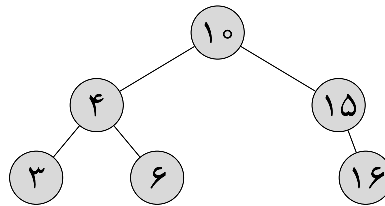
$$T(n)=\frac{۱}{n+۱}\binom{۲n}{n} \qquad (۱۱.۵)$$

که عدد nام کاتالان نامیده میشود.
مسائل مختلفی وجود دارند که راه حل اعداد کاتالان برای آنها کاربرد دارد. به عنوان نمونه میتوان به موارد زیر اشاره نمود: مساله جایگشتهای پشته، پرانتزگذاری یک عبارت ریاضی با $n+۱$ عامل، مثلث بندی یک چند ضلعی محدب.

۵.۵ درخت جستجوي دودويي

درخت جستجوی دودویی[۱۳] يك درخت دودويي است که خواص زیر را دارد: هر گره درخت يك فيلد به نام فيلد کليد (شاخص) دارد که منحصر به فرد است.
شاخص گره سمت چپ از شاخص گره والد کوچکتر است.
شاخص گره سمت راست از شاخص گره والد بزرگتر است.
در شکل ۱۴.۵ یک درخت جستجوی دودوئی با ۶ گره را مشاهده می کنید.

[۱۳] Binary Search Tree(BST)



شکل ۱۴.۵: یک درخت جستجوی دودوئی با ۶ گره

ساختار BST به گونه‌ای است که در صورت درج تصادفی عناصر در این درخت عملیات درج، حذف و جستجو را در بهترین حالت در زمان $O(logn)$ انجام میدهد. بنابراین مزیت آن در مقابل لیست نامرتب روال جستجوی آن است و مزیت آن در مقابل لیست مرتب که به کمک آرایه پیاده سازی شده است روال درج و حذف آن است. عملیات درج، حذف و جستجو در BST را در ادامه شرح خواهیم داد.

۱.۵.۵ جستجو در BST

عملیات جستجو گرهی از از درخت را که حاوی کلید x باشد را مییابد. این کار با یک پیمایش ساده با شروع از ریشه درخت انجام میشود. اگر کلید ورودی برابر با کلید ریشه باشد یعنی جستجو موفق بوده است. اگر کلید داده شده از کلید ریشه کوچکتر باشد به سمت زیر درخت چپ پیمایش میشود و اگر کلید داده شده از کلید ریشه بزرگتر باشد به سمت زیر درخت راست پیمایش میشود. عملیات پیمایش تا رسیدن به یک گره تهی انجام میشود که نشان دهنده شکست عملیات است.

شبه‌کد ۱۶.۵: جستجو در درخت جستجوی دودویی

```
1 NODE BSTfind(tree *t, keyelement x) {
2  NODE q;
3  q=ROOT(t);
4  while (!ISEMPTY(q)) {
5  if (x==KEY(t,q)) return q;
6  if (x < KEY(t,q)) q=LEFT(q); else q=RIGHT(q);
7  }
8  return q;
9 }
```

۲.۵.۵ درج در BST

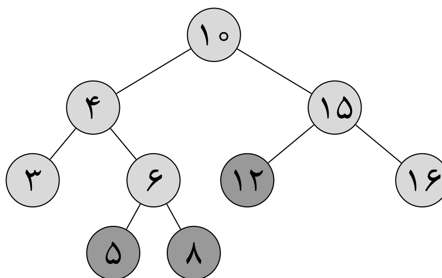
در شبه‌کد ۱۷.۵ الگوریتم درج در BST نوشته شده است. همانگونه که در شبه‌کد ۱۷.۵ مییبینید. برای درج در BST اگر درخت تهی باشد گره جدید ایجاد میشود و

به عنوان ریشه درخت انتخاب میشود. اگر درخت تهی نباشد باید به دنبال یک والد مناسب برای گره جدید باشیم. این عملیات در خط ۸ الی ۹ روال درج مشخص شده است. اگر کلید ورودی(x) از کلید ریشه درخت کوچکتر باشد به سمت زیر درخت چپ پیمایش میشود و اگر کلید ورودی از کلید ریشه بزرگتر باشد به سمت زیر درخت راست پیمایش میشود. در هر مرحله گره والد در متغییر $parent$ ذخیره میشود. این کار تا رسیدن به یک گره تهی انجام میشود. در ادامه با ایجاد یک گره جدید در خط ۱۵ تا ۱۸ گره جدید با توجه به کلید به عنوان فرزند چپ یا راست گره والد به درخت اضافه میشود. عملیات درج آدرس گره درج شده را به عنوان خروجی بر میگرداند.

شبه‌کد ۱۷.۵: درج در درخت جستجوی دودویی

```
1  NODE BSTinsert(tree *t, dataelement x) {
2   NODE q, parent=0;
3
4   if (ISNULL(t,ROOT(t))) {
5          ROOT(t)=BTInsert(t, x, 0, RIGHTCHILD);
6          return  ROOT(t);
7   }
8
9   q=ROOT(t);
10  while (!ISNULL(t,q)) {
11  parent= q;
12         if (x==KEY(t,q)) return NULL;
13         if (x < KEY(t,q)) q=LEFT(q); else q=RIGHT(q);
14  }
15
16  if ( x < KEY(t,parent))
17         return BTInsert(t, x, parent, LEFTCHILD);
18  else
19         return BTInsert(t, x, parent, RIGHTCHILD);
20 }
```

مثال ۲.۵:می‌خواهیم درج کلید ۱۲، ۵، ۸ در درخت شکل ۱۴.۵ را انجام دهیم. با توجه به شکل ۱۴.۵ بعد از ایجاد گره با کلید ۱۲، گره مربوطه به عنوان فرزند چپ گره با کلید ۱۵ درج می‌شود. گره ۵ به عنوان فرزند چپ گره۶ و گره ۸ به عنوان فرزند راست گره ۶ در درخت ۱۴.۵ درج می‌شوند. نتیجه نهایی در شکل ۱۵.۵ به تصویر کشیده شده است.



شکل ۱۵.۵: درج کلید ۱۲ در درخت جستجوی شکل ۱۴.۵

۳.۵.۵ حذف یک گره از BST

برای حذف یک گره به یکی از دو روال $BSTbiggest$و $BSTsmallest$ نیاز می‌باشد. روال $BSTbiggest(i)$، گره حاوی بزرگترین کلید در درخت BST به ریشه i را می‌یابد. این گره سمت راست ترین فرزند درخت است. پس با یک پیمایش به سمت راست درخت این گره یافت می‌شود. روال $BSTsmallest(i)$، گره حاوی کوچکترین کلید در درخت BST به ریشه i را می‌یابد. این گره سمت چپ ترین فرزند درخت است. پس با یک پیمایش به سمت چپ درخت این گره یافت می‌شود. در شبه‌کد ۱۸.۵ روالهای $BSTbiggest$و $BSTsmallest$ مشخص شده است.

شبه‌کد ۱۸.۵: یافتن عنصر ماکزیمم و مینیمم

```
1  NODE BSTbiggest(tree * t, NODE ix) {
2   while (ISVALID(ix) && !ISNULL(t,RIGHT(ix)))
3          ix=RIGHT(ix);
4   return ix;
5  }
6
7  NODE BSTsmallest(tree *t,NODE ix) {
8   while (ISVALID(ix) && !ISNULL(t,LEFT(ix)))
9          ix=LEFT(ix);
10  return ix;
11 }
```

برای حذف یک گره شرایط زیر وجود دارد:

- اگر گره مورد نظر یک برگ باشد به سادگي حذف میشود.
- اگر گره مورد نظر از درجه یک باشد باز هم بسادگی حذف میشود.
- در غیر اینصورت، اگر گره حذفی از درجه ۲ باشد، یکی از عملیات های زیر را انجام میدهیم: کوچکترین عنصر از زیر درخت سمت راست را پیدا کرده و آن

را با گره مورد نظر عوض میکنیم و عمل حذف را از محل جدید مجددا تکرار میکنیم.
بزرگترین عنصر در زیر درخت سمت چپ را پیدا کرده و آن را با گره مورد نظر عوض مي کنیم، سپس عمل حذف را از محل جدید مجددا تکرار میکنیم.
در هر يك از اين دو روش با حذف گره ها، درخت هاي يكساني بدست نمي آيد.
شبه‌کد ۱۹.۵ عملیات حذف را نشان می‌دهد.

شبه‌کد ۱۹.۵: حذف از درخت جستجوی دودویی

```
1  int BSTremove(tree *t,NODE ix) {
2   if (ISNULL(t,ix)) return 0;
3
4   NODE y=PARENT(ix);
5   if (ISNULL(t,LEFT(ix)) && ISNULL(t,RIGHT(ix))) {
6           if (ISNULL(t,y)) ROOT(t)=NULL;
7
8   BTfree(ix);
9   return 1;
10  }
11
12  if (ISNULL(t,LEFT(ix)) || ISNULL(t,RIGHT(ix))){
13         NODE child=(ISNULL(t,LEFT(ix)))? RIGHT(ix): LEFT(ix);
14         if (ISNULL(t,y)) ROOT(t)=RIGHT(ix);
15         else {
16          if (LEFT(y)==ix) LEFT(y)=child; else RIGHT(y)=child;
17         }
18         BTfree(ix);
19         return 1;
20  }
21
22  NODE q=BSTsmallest(t,RIGHT(ix));
23  KEY(t,ix)=KEY(t,q);
24  DATA(ix)=DATA(q);
25
26  return BSTremove(t,q);
27 }
```

کوچکترین گره در زیر درخت سمت راست گرهی از درجه یک یا از درجه صفر است، چون اگر قرار باشد از درجه ۲ باشد پس یک گره کوچکتر از به عنوان فرزند چپ وجود خواهد داشت. و اگر از درجه یک باشد مسلما فرزند چپ نخواهد داشت. بنابراین در صورتی که گره از درجه ۲ باشد در مرحله بعدی حذف با حذف یک گره از درجه

صفر یا یک روبرو خواهیم بود و فراخوانی مجدد روال حذف یک بار ممکن است. با توجه به وجود روال $BSTsmallest$ در بدترین حالت درجه پیچیدگی الگوریتم حذف به ارتفاع درخت بستگی دارد.

۶.۵ هرم

هرم[۱۴] یک درخت کاملی است که مقدار عنصر کلیدی گرههای فرزند از مقدار عنصر کلیدی گره والد بیشتر یا مساوی (کمتر یا مساوی) باشد. اگر از درخت دودویی برای پیاده سازی هرم استفاده کنیم با هرم دودویی روبرو هستیم. مطابق تعریف دو نوع هرم خواهیم داشت: هرم بیشینه و هرم کمینه. با توجه با ساختار یکسان این دو نوع هرم، هرم بیشینه را مورد بررسی قرار میدهیم. در شبه‌کد ۱۶.۵ یک هرم بیشنه مشاهده می‌شود.

از آنجاییکه در هرم بیشینه همیشه گره با بزرگترین کلید در ریشه قرار دارد، هرم ساختار مناسبی برای پیاده سازی صف اولویت[۱۵] می‌باشد. عملیات قابل انجام روی هرم عبارتند از:

- $heapCreate(t, n)$ از درخت t با n عنصر یک هرم ایجاد می‌کند.
- $heapIncreaseKey(i, k)$ مقدار کلید گره i را با مقدار جدید k که بزرگتر از مقدار کلید قبلی گره i است تغییر میدهد.
- $heapDecreaseKey(i, k)$ مقدار کلید گره i را با مقدار جدید k که کوچکتر از مقدار کلید قبلی گره i است تغییر می‌دهد.
- $heapInsert(h, x)$ آیتم با عنصر کلیدی x را در هرم h درج می‌کند.
- $heapDelete(h)$ عمل حذف از هرم h را انجام میدهد. عمل حذف همیشه از ریشه انجام می‌شود.
- $heapSize(h)$ اندازه هرم بر حسب تعداد عناصر را مشخص می‌کند. این قابلیت می‌تواند توسط ماکروی زیر برای هرم پیاده سازی شود.

```
1 #define heapSize(h)    (h)->count
```

۱.۶.۵ ایجاد هرم بیشینه

فرض کنید تعدادی کلید داده شده است، هدف ساخت یک هرم بیشینه با کلیدهای موجود است. با فرض این که اطلاعات گرههای درخت در یک آرایه نگهداری شود، لذا دستیابی به گرههای درخت به کمک اندیس امکان پذیر خواهد بود. اگر i اندیس

[۱۴] Heap
[۱۵] Priority Queue

یک گره از درخت باشد اندیس گره والد خواهد بود. و ریشه در اندیس یک ذخیره شده است. نخست عملیات $Heapify$ را به صورت زیر تعریف می‌کنیم: $Heapify$ شماره یک گره را دریافت نموده و در صورتی که کلید آن از کلید فرزندی که دارای کلید بزرگتر است(m)، بزرگتر باشد عمل جابجایی را انجام می‌دهد. این عملیات برای فرزند جابجا شده نیز تا رسیدن به یک برگ ادامه پیدا می‌کند. برای ساخت هرم بیشنه عملیات $Heapify$ را برای گره‌های درخت مطابق شبه‌کد ۲۰.۵ انجام می‌دهیم که مشاهده می‌کنید.

شبه‌کد ۲۰.۵: عملیات $Heapify$

```
1  void Heapify(tree *h,int i) {
2          if (ISNULL(h,LEFT(i))) return;
3          NODE m=LEFT(i);
4
5          if (!ISNULL(h,RIGHT(i)) && KEY(h,RIGHT(i)) > KEY(h,m))
6                   m=RIGHT(i);
7
8          if (KEY(h,i) < KEY(h,m)) {
9                  SWAP(&DATA(t,i),&DATA(t,m));
10                 Heapify(t,m);
11         }
12 }
```

شبه‌کد ۲۱.۵: تبدیل آرایه به یک هرم

```
1  tree * heapCreate(ITEM a[],int n) {
2          NODE i;
3          tree *t=BTCreate();
4          if (!t) return t;
5
6          for(i=0; i<n; i++)
7             DATA(t,i+1)=a[i];
8          COUNT(t)=n;
9          ROOT(t) =1;
10
11         for(i=n/2;i>=1; i--)
12                 Heapify(t,i);
13         return t;
14 }
```

روال $heapCreate$ در شبه‌کد ۲۱.۵فرض کرده است که n کلید قبلا در آرایه از اندیس یک تا n ذخیره شده است و روال $Heapify$ را از اندیس وسط تا اندیس اول

آرایه فراخوانی می‌کند. چون از اندیس $\lfloor\frac{n}{۲}\rfloor + ۱$ تا n گرههای برگ می باشند و نیازی به عمل $Heapify$ برای آنان نیست. به عبارت دیگر با توجه به کامل بودن هیپ، آخرین گره درخت از درجه یک یک دو در اندیس $\lfloor\frac{n}{۲}\rfloor$ واقع است.

مثال ۳.۵: آرایه‌ای از اعداد داده شده است، آن را به یک هرم بیشینه تبدیل نمایید.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	15	19	25	32	14	12	18	36	42	56	65	74	13

حل: هرم دارای ۱۴ عنصر است بنابراین عمل $Heapify$ را از عنصر ۷ تا ۱ انجام میدهیم:

$Heapify(۷)$ عنصر هفتم با عنصر چهاردهم جابجا میشود.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	15	19	25	32	14	13	18	36	42	56	65	74	12

$Heapify(۶)$

1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	15	19	25	32	74	13	18	36	42	56	65	14	12

$Heapify(۵)$

1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	15	19	25	56	74	13	18	36	42	32	65	14	12

$Heapify(۴)$

1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	15	19	36	56	74	13	18	25	42	32	65	14	12

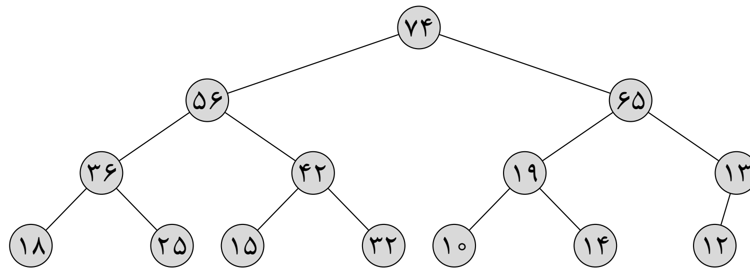
$Heapify(۳)$

1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	15	74	36	56	65	13	18	25	42	32	19	14	12

$Heapify(۲)$

1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	56	74	36	42	65	13	18	25	15	32	19	14	12

$Heapify(۱)$



شکل ۱۶.۵: هرم بیشینه مربوط به مثال ۱.۶.۵

1	2	3	4	5	6	7	8	9	10	11	12	13	14
74	56	65	36	42	19	13	18	25	15	32	10	14	12

در نهایت درخت شکل ۱۶.۵ را خواهیم داشت:

۲.۶.۵ درج در هرم بیشینه

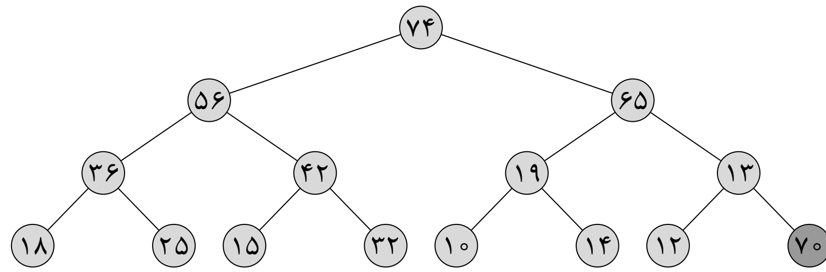
کلید جدید در آخرین محل هرم درج میشود سپس اگر کلید وارد شده از کلید والد بزرگتر باشد عمل جابجایی انجام میشود. این مقایسه تا رسیدن به گره ریشه انجام میگردد.

شبه‌کد ۲۲.۵: عملیات درج در هرم

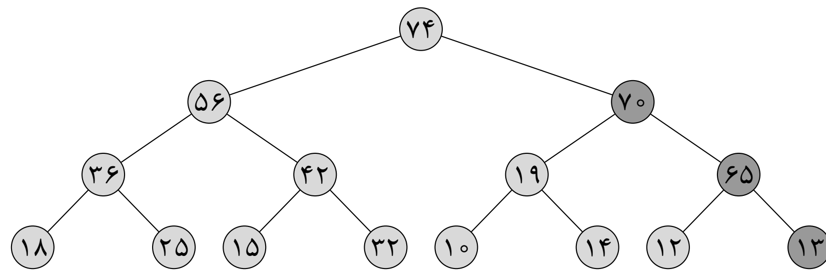
```
1 void heapInsert(tree *h, ITEM x) {
2         i=heapSize(h)+1;
3         DATA(h,i)=x;
4         KEY(h,i)=x;
5         while (i>1 && KEY(h,i) > KEY(h,PARENT(i)))) {
6                 SWAP(DATA(h,i),DATA(h,PARENT(i)));
7                 i=PARENT(h,i);
8         }
9 }
```

مثال ۴.۵:مثال: فرض کنیم کلید ۷۰ در هرم مثال قبل اضافه شود در آنصورت خواهیم داشت: کلید ۷۰ در آخرین محل آرایه که در اینجا ۱۵ است، ذخیره میشود. که در شکل ۱۷.۵ مشخص شده است. اما در این حالت خاصیت هرم‌ماکزیمم برقرار نیست لذا نیاز به جابجایی می‌باشد.

- حال اندیس ۱۵ با اندیس والدش یعنی ۷ مقایسه می‌شود. چون بزرگتر است جابجا میشود.
- حال اندیس ۷ با اندیس والدش یعنی ۳ مقایسه می‌شود. چون بزرگتر است جابجا میشود.



شکل ۱۷.۵: درج کلید ۷۰ در هرم شکل ۱۶.۵ قبل از اعمال جابجایی



شکل ۱۸.۵: درج کلید ۷۰ در هرم شکل ۱۶.۵ بعد از انجام جابجایی

- **حال اندیس ۳ با اندیس یک مقایسه میشود، چون کوچکتر است هیچ اتفاقی نمی‌افتد و تمام.**

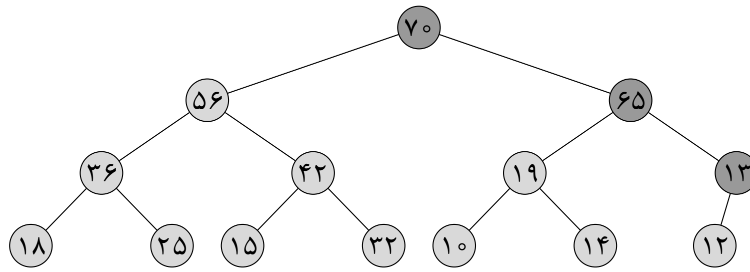
نتیجه درخت شکل ۱۸.۵ خواهد شد:

۳.۶.۵ حذف از هرم بیشینه

حذف هرم از ریشه انجام میشود. ابتدا ریشه با آخرین عنصر جابجا شده و حذف میشود. سپس عمل $Heapify$ **روی گره ریشه انجام میشود.**

شبه‌کد ۲۳.۵: عملیات حذف از هرم

```
1 ITEM heapDelete(tree *h) {
2         ITEM temp;
3         NODE i=heapSize(h);
4         temp=DATA(h,1);
5         SWAP(&DATA(h,ROOT(h)), &DATA(h,i));
6         DATA(h,i)=NIL;  // remove last child
7         COUNT(h)--;
8         Heapify(h,ROOT(h));
9         return temp;
```



شکل ۱۹.۵: حذف از هرم شکل ۱۸.۵، حذف از ریشه انجام می‌شود

```
10 }
```

مثال ۵.۵:فرض کنیم بخواهیم عمل حذف را روی درخت شکل ۱۸.۵ انجام دهیم، واضح است که ریشه باید حذف گردد، لذا خواهیم داشت:

- ابتدا ریشه با آخرین گره جابجا شده و حذف میشود.
- حال عمل $Heapify(ROOT(h))$ انجام میشود. گره در اندیس یک یعنی ۱۳ با گره در اندیس ۳ یعنی ۷۰ جابجا می‌شود و گره ۷۰ در ریشه قرار می‌گیرد. چون گره ۷۰ بزرگتر است.
- سپس گره ۱۳ با گره ۶۵ جابجا میشود.
- چون گره ۱۳ از گره ۱۲ بزرگتر است دیگر نیاز به جابجایی نیست و تمام.

شکل ۱۹.۵ نتیجه عملیات حذف می‌باشد.

۴.۶.۵ افزایش کلید

در بعضی از کاربردها ممکن است لازم باشد کلید یک گره افزایش پیدا کند. روال **heapIncreaseKey** این کار را انجام می‌دهد. بعد از افزایش کلید لازم است به سمت ریشه حرکت کرده و در صورت نیاز عمل جابجایی گره‌ها را انجام دهیم. این عملیات در کد زیر مشخص شده است. پیچیدگی زمانی روال **heapIncreaseKey** از درجه **O(log(n))** است.

شبه‌کد ۲۴.۵: عملیات افزایش کلید در هرم

```
1 int heapIncreaseKey(tree *h,NODE i, int k) {
2         if (k < KEY(h,i))       {
```

```
3                 printf("error");
4                 return 0; //unsuccessfull
5         }
6         KEY(h,i)=k;
7         while (i>1 && KEY(h,PARENT(i)) < KEY(h,i)) {
8                 SWAP(DATA(h,i), DATA(h,PARENTNT(i));
9         }
10        return 1; // successfull
11 }
```

۵.۶.۵ مرتب سازی هرمی

مرتب سازی هرمی یکی از روشهای مرتب سازی اطلاعات می‌باشد. در روش مرتب سازی هرمی ابتدا از روی ورودی نامرتب که به صورت آرایه می‌باشد یک هرم بیشینه(یا کمینه) ساخته می‌شود. سپس عملیات حذف از هرم انجام شده و داده‌ها به صورت مرتب در خروجی مرتب قرار می‌گیرند. از آنجاییکه در هرم بیشنه(کمینه) در هر مرحله حذف، بزرگترین(کوچکترین) عنصر خارج می‌شود. لذا مرتب سازی به صورت نزولی(صعودی) می‌باشد.
عملیات مرتب سازی هرمی را در کد ۲۵.۵ مشاهده می‌کنید. ورودی نامرتب در آرایه a ذخیره شده است. پارامتر n تعداد عناصر آرایه را نشان می‌دهد. در ابتدا یک هرم از روی آرایه نامرتب ورودی ایجاد می‌شود. سپس همه عناصر از هرم خارج شده و به صورت مرتب شده در همان آرایه ورودی یعنی a قرار داده می‌شوند.

شبه‌کد ۲۵.۵: مرتب سازی هرمی

```
1  ITEM * heapSort(ITEM a[], int n) {
2         tree* h=heapCreate(a,n);
3         ITEM x;
4         int k=0;
5         while (heapSize(h)>0) {
6                 x=heapDelete(h);
7                 a[k++]=x;
8         }
9         return a;
10 }
```

۷.۵ تمرینات فصل

تمرین ۵.۱: الگوریتمی بنویسید که تعداد گره هاي برگ يك درخت دودويي را مشخص کند. زمان اجرايي این الگوریتم چقدر است؟ (راهنمايي: تعداد برگ هاي دودويي: تعداد برگ هاي زير درخت سمت راست + تعداد برگ هاي زير درخت سمت چپ) ■

تمرین ۵.۲: الگوریتمی به نام *swapChilds* بنویسید که در يك درخت دودويي، فرزندان راست و چپ هر گره را عوض نماید. ■

تمرین ۵.۳: تابعي به زبان C بنویسید که اگر کلمه‌اي حالت آينه‌اي داشته باشد مقدار *true* و اگر در غير اين صورت بود *false* را برگشت دهد. (با استفاده از پشته). ■

تمرین ۵.۴: کلید های زیر از چپ به راست به یک درخت جستجوی دودویی اضافه می شود. (در ابتدا درخت تهی است) ۵۰ ، ۳۵ ، ۸۰ ، ۴۰ ، ۲۵ ، ۷۵ ، ۳۲ ، ۹۶

- الف- درخت را بعد از درج هر یک از کلید ها رسم کنید.
- ب- در درخت نهایی پیمایش *inorder* را بنویسید.
- ج- در درخت نهایی پیمایش *Preorder* را بنویسید.
- د- در درخت نهایی پیمایش *levelorder* را بنویسید.

■

تمرین ۵.۵: یک درخت دودویی را دریک آرایه به صورت زیر ذخیره شده است، این درخت را ترسیم کنید. ■

تمرین ۵.۶: در يك درخت دودويي پر به عمق ۱۰، تعداد گره ها از درجه دو را به دست آورید. ■

تمرین ۵.۷: مدل غير بازگشتي الگوريتم درج يك گره در يك درخت دودويي جستجو را بنویسید. ■

تمرین ۵.۸: فرض کنید که یک درخت جستجو ۱۰۰۰۰۰ کلید را ذخیره کرده

است. کدام گزینه بدترین حالت را برای ارتفاع درخت T ممکن است بسازد.

۱. T یک درخت قرمز-مشکی باشد.

۲. T یک درخت AVL باشد.

۳. T یک درخت ۴-۳-۲ باشد.

۴. T یک درخت جستجوی دودویی باشد.

■

تمرین ۹.۵: یک شرکت مخابراتی دارای ۲۵۶ میلیون مشتری است. فرهنگ تلفن این شرکت بدلیل حجیم بودن آن قابل چاپ شدن نیست و در صورت چاپ شدن گران و غیر قابل استفاده است. برای حل این مشکل این شرکت تصمیم به ذخیره سازی و ارائه یک سیستم کامپیوتری آنلاین نمود و مهندسین نرم افزار آن سعی در طراحی یک ساختمان داده مناسب برای ذخیره اطلاعات مشتریان می باشند. فرض کنید که کامپیوتر این شرکت می تواند عمل مقایسه دو نام را در زمان یک میکرو ثانیه انجام دهد.

- یکی از مهندسین پیشنهاد استفاده از یک لیست پیوندی مرتب نشده را داده است. با این پیاده سازی تخمینی از بهترین و بدترین حالت برای عملیات درج و جستجو ارائه کنید.

- یک مهندس دیگر پیشنهاد استفاده از درخت را داده است. با این پیاده سازی تخمینی از بهترین و بدترین حالت برای عملیات درج و جستجو ارائه کنید.

- مهندس سوم پیشنهاد استفاده از آرایه مرتب شده را داده است.

آیا این پیشنهادها مناسب هستند؟ ■

تمرین ۱۰.۵: به طور خلاصه صحیح یا غلط بودن جملات زیر را مشخص نمایید.

- در یک هرم ، تفاوت ارتفاع گره های زیر درخت چپ و راست حداکثر یک است.

- بهترین حالت زمان اجرای الگوریتم مرتب سازی حبابی $O(n)$ است.

- در بهترین حالت زمان اجرای الگوریتم $mergesort$ عبارت است از $O(n)$.

- در بدترین حالت پیچیدگی زمانی الگوریتم مرتب سازی سریع، $O(n^2)$ است.

- **در بدترین حالت درج در درخت AVL از مرتبه $O(n)$ می باشد.**

■

ضمیمه

۸.۵ درخت دودویی به کمک آرایه

شبه‌کد ۲۶.۵: درخت دودویی به کمک آرایه

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define         ITEM            int
5  #define         keyelement      int
6  #define         MAX             256
7  #define         NODE            int
8
9  struct tree{
10         ITEM    nodes[MAX];
11         int     root;
12         int     count;
13 };
14 typedef struct tree tree;
15
16 #define ROOT(T)          (T)->root
17 #define LEFT(i)          (2*i)
18 #define RIGHT(i)         (2*i+1)
19 #define PARENT(i)        (i/2)
20
21 #define DATA(T,i)        (T)->nodes[i]
22 #define KEY(T,i)         (T)->nodes[i]
23 #define COUNT(T)         (T)->count
24 #define ISVALID(i)       (i>0 && i<MAX)
25 #define ISNULL(T,i)      (!ISVALID(i) || (T)->nodes[i]== 0)
26
27 #define LEFTCHILD        0
28 #define RIGHTCHILD       1
```

```
29 #define NIL             0
30
31 void SWAP(ITEM *a, ITEM *b) {
32  ITEM t=*a;
33  *a=*b;
34  *b=t;
35 }
36
37 tree * BTCreate() {
38  tree *t;
39  t= (tree *) malloc(sizeof(struct tree));
40  if(t!=NULL){
41          int i;
42          for(i=0; i<MAX; i++) t->nodes[i]=0;
43          ROOT(t)=1;
44          COUNT(t)=0;
45  }
46  return t;
47 }
```

۹.۵ ساختار هرم دودویی

شبه‌کد ۲۷.۵: ساختار هرم دودویی

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define  heapSize(h)    (h)->count
5
6  void Heapify(tree *h,int i) {
7          if (ISNULL(h,LEFT(i))) return;
8          NODE m=LEFT(i);
9
10         if (!ISNULL(h,RIGHT(i)) && KEY(h,RIGHT(i)) > KEY(h,m))
11                  m=RIGHT(i);
12
13         if (KEY(h,i) < KEY(h,m)) {
14                 SWAP(&DATA(h,i),&DATA(h,m));
15                 Heapify(h,m);
16         }
17 }
18
19 tree * heapCreate(ITEM a[],int n) {
20         NODE i;
21         tree *t=BTCreate();
22         if (!t) {
23                 printf("\nNot␣Enough␣memory!.\n");
24                 return t;
25         }
26
27         for(i=0; i<n; i++)
28            DATA(t,i+1)=a[i];
29         COUNT(t)=n;
30         ROOT(t) =1;
31
32         for(i=n/2;i>=1; i--)
33                 Heapify(t,i);
34         return t;
35 }
36
37 ITEM heapDelete(tree *h) {
38         ITEM temp;
```

```
39          NODE i=heapSize(h);
40          temp=DATA(h,1);
41          SWAP(&DATA(h,ROOT(h)), &DATA(h,i));
42          DATA(h,i)=NIL;  // remove last child
43          COUNT(h)--;
44          Heapify(h,ROOT(h));
45          return temp;
46  }
```

۱۰.۵ مرتب‌سازی هرمی

شبه‌کد ۲۸.۵: پیاده‌سازی مرتب سازی هرمی

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "bintree.h"
4  #include "heap.h"
5
6  ITEM * heapSort(ITEM a[], int n) {
7          tree* h=heapCreate(a,n);
8          ITEM x;
9          int k=0;
10         while (heapSize(h)>0) {
11                 x=heapDelete(h);
12                 a[k++]=x;
13         }
14         return a;
15 }
16
17 int  ar[MAX];
18 int  n;
19
20 int input() {
21   int x;
22   n=0;
23   scanf("%d", &x);
24   while (x!=0) {
25       ar[n++]=x;
26       scanf("%d", &x);
27   }
28   return n;
29 }
30
31 void output() {
32   int i;
33   for(i=0; i<n; i++)
34     printf("%4d", ar[i]);
35   printf("\n");
36 }
37
38 int main() {
```

```
39      tree * heap;
40
41      input();
42      heapSort( ar, 10);
43      output();
44
45      return 0;
46  }
```

*** پایان ***