

مستندات پروژه عملی دوم پینتوس

استاد: دکتر شهاب‌الدّین نبوی

اعضای گروه:

محمد خدّام

حامد خادمی خالدی

متین زیودار

امیر حلاجی

بهار 1400

[چکیده 3](#_Toc74139220)

[ساختمان داده‌ها و الگوریتم‌ها 3](#_Toc74139221)

[ساختمان داده‌ی Htree 3](#_Toc74139222)

[کاوشگر هش(Hash Probe) 4](#_Toc74139223)

[جدول هش(Hash table) 5](#_Toc74139224)

[اعمال تغییرات جدید 6](#_Toc74139225)

[ساختار btable 6](#_Toc74139226)

[تابع btCreate 6](#_Toc74139227)

[تابع btDestroy 7](#_Toc74139228)

[تابع searchKey 8](#_Toc74139229)

[تابع btSearch 9](#_Toc74139230)

[تابع btInsertInternal 10](#_Toc74139231)

[تابع btInsert 12](#_Toc74139232)

[ساختار Htable 13](#_Toc74139233)

[ساختار htable node 13](#_Toc74139234)

[تابع htCreate 14](#_Toc74139235)

[تابع htDestroy 15](#_Toc74139236)

[تابع htSearch 16](#_Toc74139237)

[تابع htInsert 17](#_Toc74139238)

# چکیده

در سال‌های اخیر، طرّاحان فایل سیستم‌، متمایل به استفاده از Btree و گونه‌های مختلف راجع به آن برای از بین بردن bottleneckهای در عملکرد دایرکتوری شده‌اند. ایده‌ی اضافه‌کردن دایرکتوری Btree به فایل سیستم Ext2 لینوکس خیلی مورد بحث قرار گرفته‌؛ امّا هیچ‌گاه به مرحله‌ی پیاده‌سازی نرسیده است و این بیشتر به خاطر بیزاری و نفرت توسعه‌دهندگان نسبت به پیچیدگی کار است و نه به خاطر تنبلی‌شان!

# ساختمان داده‌ها و الگوریتم‌ها

## ساختمان داده‌ی Htree

یک بیت flag در inode دایرکتوری، به ما نشان می‌دهد که آیا این دایرکتوری index شده‌است یا خیر. اگر بیت flag ما تعیین‌شده باشد(اصطلاحاً set شده‌باشد.)، آنگاه، اولین بلاک دایرکتوری به عنوان root از Htree شناخته می‌شود.

Root فهرست Htree، به عنوان اولین بلاک از فایل دایرکتوری شناخته می‌شود. برگ‌های Htree، بلاک‌های نرمال Ext هستند که به صورت مستقیم و یا غیر مستقیم، به root ارجاع داده می‌شوند. احتمال استفاده‌ از اشاره‌گرهای فیزیکی به دلیل سودمندی بیشتر است؛ ولی به خاطر نیازمندی شدید به پیوستگی‌سازی‌ها برای مدیریت اشاره‌گرهایی که در بسیاری از قسمت‌های Ex2 در بیرون مدیریت کد دایرکتوری هستند، این ایده و ترفند هیچ وقت به مرحله‌ی استفاده نرسید. خوشبخانه، به خاطر تغییرات اخیر لینوکس به سمت فهرست سازی منطقی(logical indexing)، بلاک‌های اشاره‌گر منطقی دیگر سودمندی و بازدهی کمتری نسبت به نوع فیزیکی ندارند.

یک htree به جای استفاده از اسم‌ها، از کلیدهایی استفاده می‌کند که در واقع همان hash اسامی هستند. هر کلید هَش، لزوماً به یک دایرکتوری منفرد و خاصّی ارجاع نمی‌دهد؛ بلکه به یک محدوده‌ای از ورودی‌ها(entry) که داخل یک بلاک منفرد برگ(single leaf block) مرتب‌سازی شده‌اند، ارجاع می‌دهد.

ریشه‌ی htree، همچنین شامل یک آرایه‌ای از ورودی‌ها به فرمت یکسان با سطح اول است. اشاره‌گرها، به بلاک‌های اندیسی(index blocks)، بیشتر از بلاک‌های برگ(leaf blocks) ارجاع می‌دهند و کلید‌های هش محدوده‌های پایین‌تر را به کلیدهای هشِ ارجاع داده‌شده توسط بلاک‌های اندیسی اختصاص می‌دهند.

## کاوشگر هش(Hash Probe)

اولین قدم در هر دایرکتوری نمایه شده(indexed directory)، پیدا کردن ریشه، اولین بلاک از فایل دایرکتوری، است. سپس در مرحله‌ی بعد، برای از بین بردن هر اندیس خراب و منحرف‌کننده‌ای که می‌تواند منجر به ایجاد مشکل در هسته‌ی سیستم‌عامل بشود، تعدادی تست انجام می‌شود. تشخیص هرگونه ناسازگاری منجر به بازگشتن عملیات دایرکتوری (directory operation) به سمت جستجوی خطی می‌شود. در این حالت، بیشترین شانس برای دسترسی به درایو(volume) خراب، به کاربر داده می‌شود.

## جدول هش(Hash table)

هر جدول هش نرمال، یک آرایه‌ی خطّی از bucket ها است و کلید هش، به صورت مستقیم bucketای را index می‌کند که در جستجوی آن هستیم. بنابراین، جستجو برای پیدا کردن bucket صحیح خیلی سریع می‌انجامد. یک اشکال اصلی در جدول هش این است که اندازه‌ی آن، نه خیلی بزرگ و نه خیلی کوچک باشد. جدول هشی که خیلی بزرگ باشد، منجر به اتلاف فضا و جدول هش خیلی کوچک منجر به برخوردهای مکرر و با تعداد بالا می‌شود. انتخاب یک اندازه‌ی مناسب برای جدول هش، یک مسئله‌ی چالشی است. راه‌ حل این مشکل این‌ است که اندازه‌ی جدول هش به اندازه‌ی تعداد رشته‌های موجود در جدول هش بالا برود. هنگامی که جدول هشِ ما در آستانه‌ی پر شدن قرار می‌گیرد، محتویات فعلی جدول به علّت عدم فضای کافی، به یک جدول بزرگ‌تر منتقل می‌شوند و عملیاتی تحت عنوان «هش‌ مجدد» (rehashing)، صورت می‌گیرد.

# اعمال تغییرات جدید

## ساختار btable

همانطور که مشخص است، btNode یک struct است که تعدادی فیلد دارد.

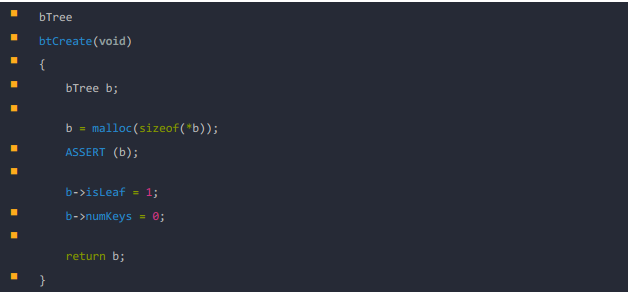
فیلد isLeaf بیان می‌کند که آیا node ما یک برگ است یا خیر.

فیلد numKeys بیان می‌کند که گره‌ی فعلی چه تعداد کلید را شامل می‌شود.

همچنین، یک آرایه از اعداد صحیح را نگهداری می‌کنیم که اندازه‌ی آن به تعداد بیشینه‌ی کلیدها است. علاوه بر این‌ها، آرایه‌ای از offset ها به همین اندازه داریم.

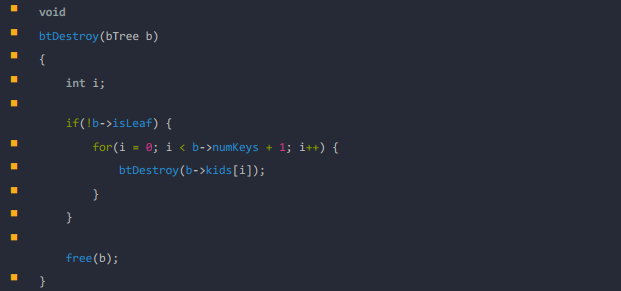
دو ساختار دیگر نیز به نام‌های entries از جنس dir\_entries و یک اشاره‌گر به نام kids از جنس btNode داریم.

## تابع btCreate



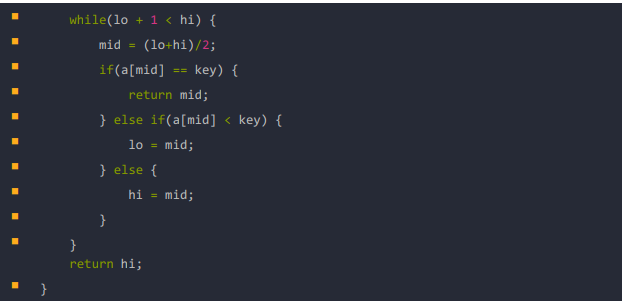
این تابع، ورودی‌ای دربافت نمی‌کند و یک نمونه از bTree را به ما در خروجی برمی‌گرداند. در ابتدای تابع، یک نمونه از bTree تعریف می‌کنیم و به اندازه‌ی آدرس آن، به آن حافظه تخصیص می‌دهیم. چون این گره، در حال حاضر کلیدی را شامل نمی‌شود، فیلد numKeys را صفر و چون برگ است، فیلد isLeaf را 1 می‌گذاریم.

## تابع btDestroy



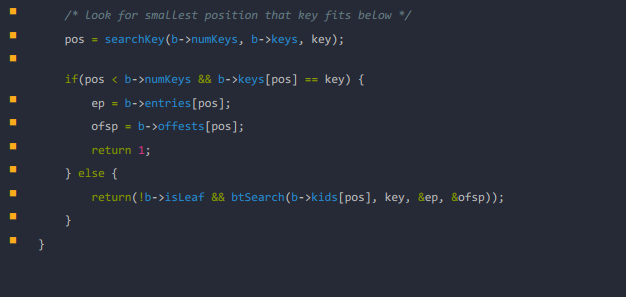
این تابع، یک ورودی از جنس bTree دریافت می‌کند و خروجی‌ای برنمی‌گرداند. نحوه‌ی کارش به این‌گونه است که در ابتدا یک i تعریف می‌کنیم و اگر b یک برگ نباشد، به اندازه‌ی تعداد کلیدهایش پیمایش می‌کنیم و بچه‌هایش را destroy می‌کنیم. پس از این‌ها در انتها، b را آزاد می‌کنیم.(free(b)).

## تابع searchKey



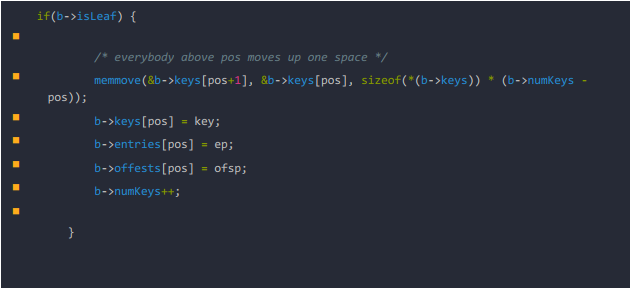
این تابع به ما یک عدد صحیح برمی‌گرداند که key ای است که به دنبال آن می‌گردیم. نحوه‌ی عملکرد آن به این صورت است که hi اندیس بالا و lo اندیس پایین است. تا زمانی که lo به hi نرسیده یعنی تا زمانی که کلید مورد نظرمان پیدا نشده است، حلقه را پیمایش می‌کنیم و هر سری اندیس وسط را برابر میانگین بالا و پایین می‌گذاریم. اگر عنصر با اندیس وسط همان کلید موردنظرمان باشد، همان مقدار اندیس mid را به عنوان خروجی تابع بر می‌گردانیم.

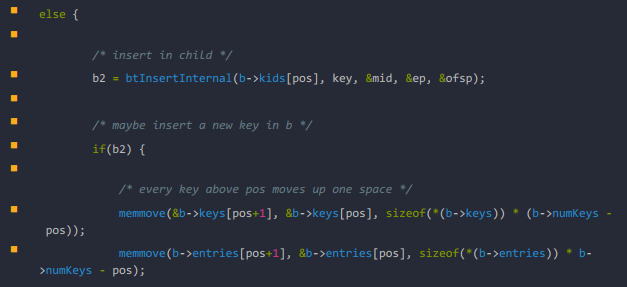
## تابع btSearch

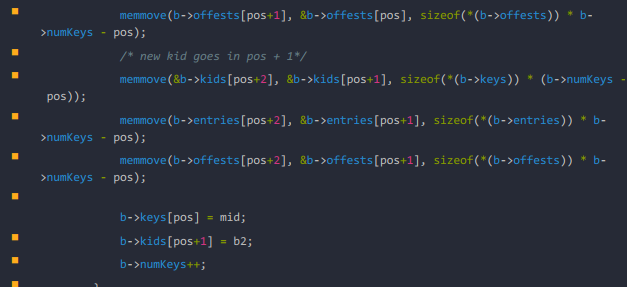


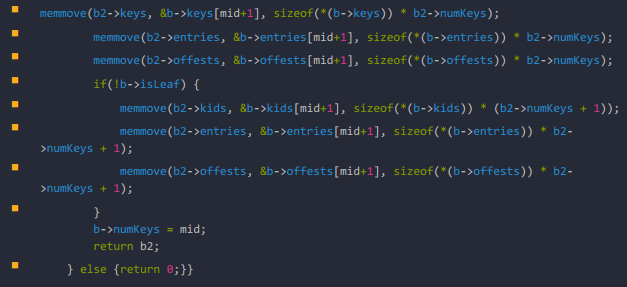
این تابع...

## تابع btInsertInternal



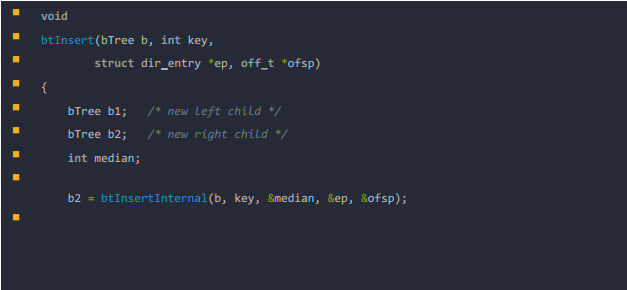


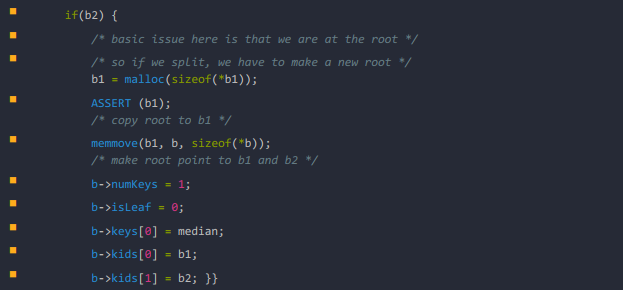




این تابع...

## تابع btInsert





این تابع...

# ساختار Htable

## ساختار htable node

هر htNode، بک struct است که دو فیلد از جنس bTree و hash دارد.

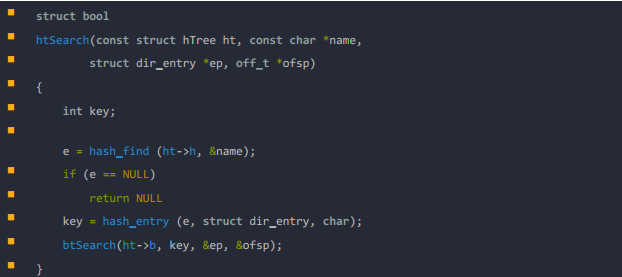
## تابع htCreate

این تابع...

## تابع htDestroy

این تابع..

## تابع htSearch



این تابع...

## تابع htInsert

این تابع...