کارشناسی ارشد شبکه های کامپیوتری

شبکه های پهن باند

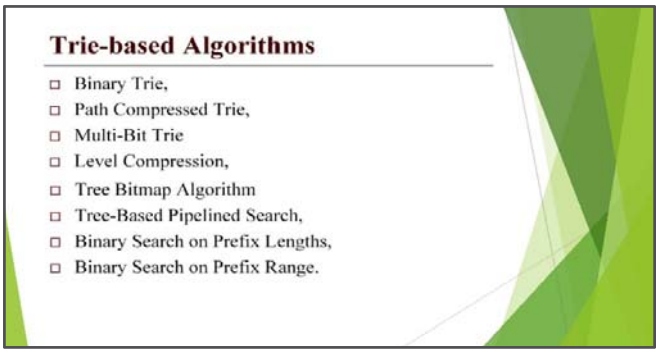
استاد:

دکتر جبار پور

نویسنده :

مهندس علیرضا روحانی نژاد

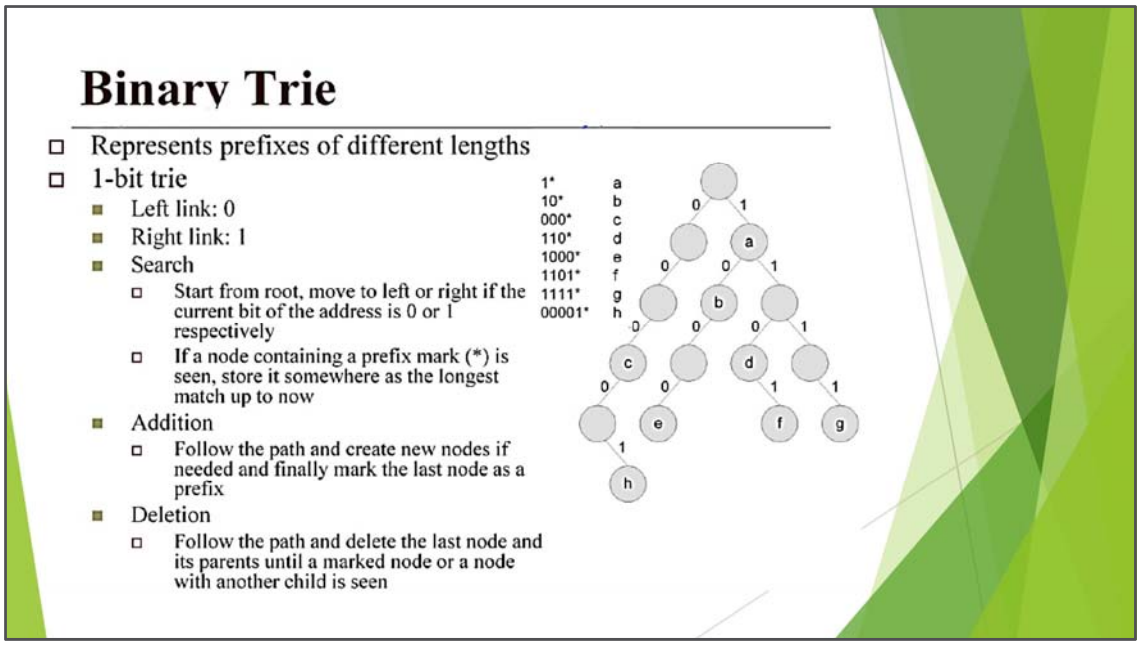
۱۳۹۷



دو روش جهت افزایش سرعت مسیریاب‌ها ذکر شد: روش سخت‌افزاری- روش نرم‌افزاری

روش‌های نرم‌افزاری برمبنای درخت هستند و به آن‌ها Tree Based Algorithm گفته می‌شود. در ادامه مباحث، روش‌های بر این مبنا را بررسی می‌کنیم.

Binary Tree: درخت دودویی: سرعت جستجو بالاست ولی حافظه مصرفی کاملا به نوع درختی که داریم استفاده می‌کنیم، بستگی دارد. می‌تواند کم یا زیاد باشد. سرعت Update هم بستگی به درخت دارد.



یکی از ساختارهای داده برای جستجو (Longest Matchها) حالت درختی است. در ساده‌ترین حالت درختی دودویی داریم که اطلاعات Prefixها را در یال ها ذخیره می‌کند. هر عدد روی یال مشخص کننده عددهایی است که در جدول مسیریابی هستند. اعداد در این درخت معادل IPهای Destination هایی هستند که ما در Routing Table داریم و a, b, c,…,h پورت نامبرهای خروجی هستند که مشخص می‌کند ما ازکدام پورت می‌توانیم بسته‌ها را به‌سمت مقصد هدایت کنیم. با

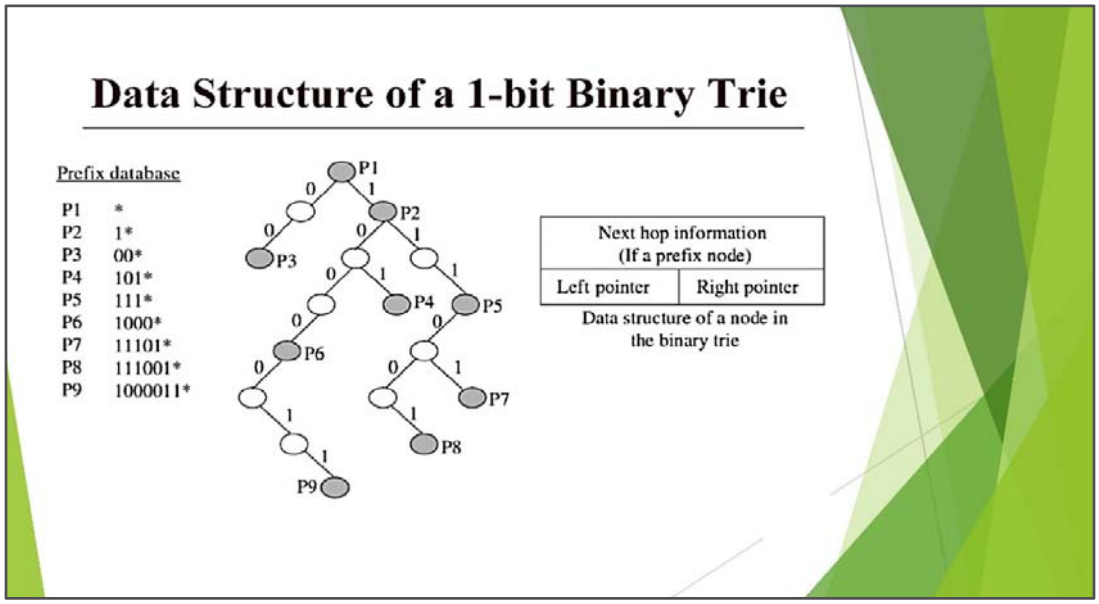
شروع از بیت پر ارزش که می‌تواند ۰ یا ۱ باشد، یکی از فرزندان ریشه درخت را برای پیش رفتن انتخاب می‌کنیم. بعد از رسیدن به آخرین بیت، اسم آن پیشوندی که در Node فعلی است را به آن اضافه می‌کنیم. مطابق با شکل، نام پیشوندها در Nodeهای غیر برگ هم قرار گیرند. الزامی نیست که همیشه در برگ‌ها بخواهیم Prefixها را قرار دهیم. مثلا پیشوند b، ۱۰ است. خود آن زیر مجموعه‌ای از حالت e است به همین دلیل امکان تطبیق آدرس پیش رو با چند پیشوند وجود دارد و تاکید می‌شود در این مورد بایستی طولانی‌ترین پیشوند را به عنوان نتیجه در نظر بگیریم.

جستجو برای این درخت بسیار ساده است:با شروع از ریشه، با توجه به صفر و یک بودن، بیت‌های آدرس را مورد جستجو قرار می‌دهیم پس از چک کردن هر بیت به فرزند چپ یا راست آن منتقل می‌شویم. حین انتقال هر جا Nodeی جلوی یک Prefix رسیدیم آن‌را به‌عنوان طولانی‌ترین تطبیق به حافظه می‌سپاریم تا به Node بزرگ برسیم یا امکان ادامه جستجو مقدور نباشد. آن‌جاست که جستجوی ما به پایان رسیده‌است.(مثال جلسه قبل)

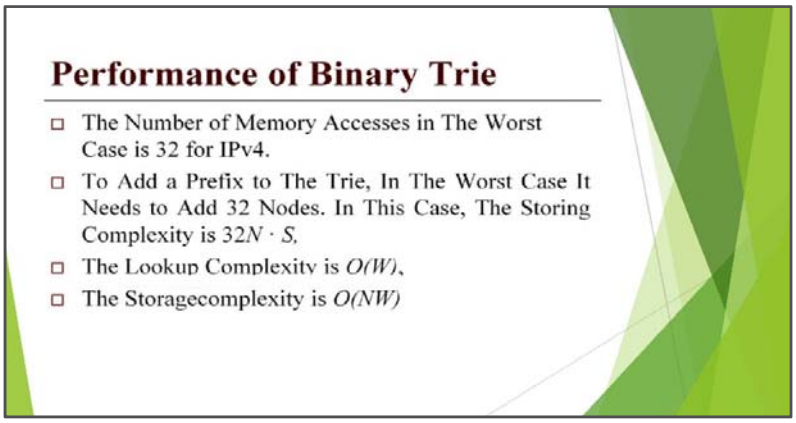
وقتی می‌خواهیم Entry جدید اضافه کنیم، باید قبل آن پیشوند جدید اضافه شود. ابتدا باید شبیه جستجوی درخت جلو برویم و مسیر موجود را تا جایی که وجود دارد، ادامه دهیم، اگر این مسیر به‌طور کامل وجود داشت، کافی است نام پیشوند را در آن Nodeای که در انتهاست ثبت کنیم در غیر این‌صورت ادامه مسیر مربوط به آن Node را در درخت ایجاد می‌کنیم و بعد از ایجاد کردن، Prefixی که مدنظرمان است را در Node پایانی ذخیره می‌کنیم.

برای حذف(Delete) اول مسیر آن پیشوند را تا انتهایش می‌رویم. اگر آخرین Node مربوط به آن مسیر دارای فرزندی بود مثل پیشوندC کافی است نام C را در درخت حذف کنیم. کار دیگری نیاز نیست چون اگر بخواهیم از C به بعد را پاک کنیم، Prefix h را ازدست خواهیم داد. اگر بخواهیم Nodeی را حذف کنیم که فرزندی ندارد باید خود آن Node را حذف کنیم و والد آن‌را هم باید حذف کنیم.

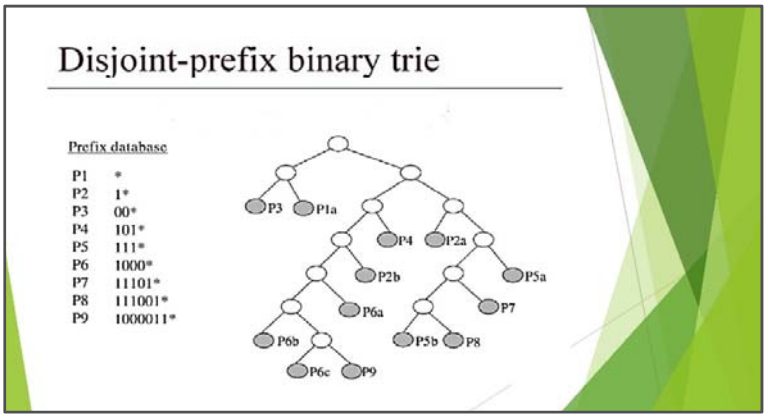
برای مثال: اگر بخواهیم h را حذف کنیم، h و Node والد را حذف کنیم به‌نحوی که C به برگ تبدیل شود.(چند مثال دقیقه ۱۱ کپچر)



درخت باینری ۱بیتی را توضیح می‌دهد. به‌این صورت است که هر گره، دارای دو اشاره‌گر چپ و راست است و اطلاعات Next Hop داخل گره ذخیره می‌شود. این‌ها اطلاعاتی است که برای حالت ۱بیتی باید داشته باشیم.

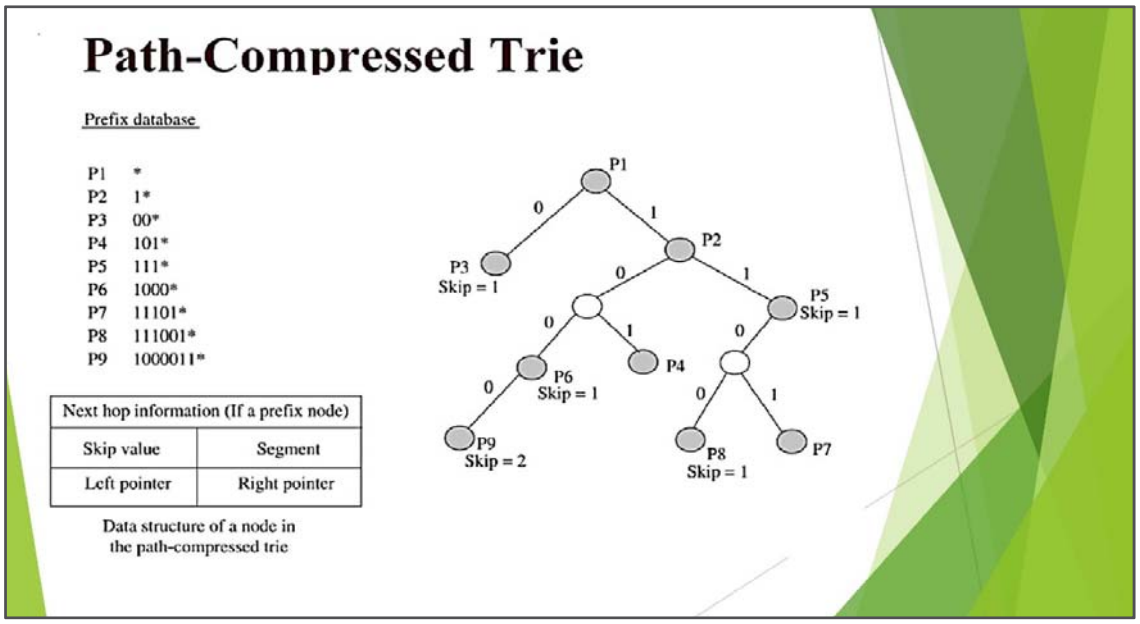


اگر بخواهیم کارایی این درخت را بررسی کنیم، گفتیم که در بدترین حالت تعداد دسترسی‌ها به حافظه برای جستجو حالتی است که عمق درخت ۳۲بیتی باشد چون 1PV4، ۳۲بیتی است و Prefixهای ما باید ۳۲بیتی باشد چون در این حالت باید ۳۲بار به حافظه رجوع کنیم. این مشخص می‌کند کدام Entry مدنظر است. برای اضافه کردن در بدترین حالت، بایستی ۳۲ Node اضافه شود. هیچ مسیری وجود نداشته باشد و مجبور شویم همه Nodeها را اضافه کنیم. اگر عمق درخت را W فرض کنیم، در این اسلاید که می‌شود طول بزرگترین Prefixمان، پیچیدگی جستجو برابر با O(W) خواهد بود و حجم مصرفی‌ها (N×W) خواهد شد به‌ازای Nodeی که وجود دارد ضربدر عمقی که خواهیم داشت. حال اگر 1PV6 مطرح شود حداکثر عمق ما به‌جای ۳۲، ۱۲۸ خواهد بود.



می‌خواهیم درخت باینری که جلسه قبل درباره‌اش حرف زدیم را تغییر دهیم تا حافظه مصرفی‌اش را کم کنیم و تا حد امکان سرعت جستجو را افزایش دهیم. هرچقدر حافظه مصرفی کم شود، در حقیقت زمان نیاز به دسترسی حافظه هم کمتر خواهد بود. درخت اسلاید همان درخت دودویی است که «Disjoint» شده‌است. در حالت Disjoint، Nodeهای میانی دیگر اطلاعات Next Hop را در خودشان نگهداری نمی‌کنند فقط گره‌های برگ هستند که اطلاعات Next Hop را ذخیره می‌کند. پس در گره‌های میانی مقداری نداریم و گره‌های برگ هستند که مقدار Next Hop را به‌خودشان گرفته‌اند. برای ایجاد این درخت، اول به Nodeهایی که یک فرزند دارند، یک برگ اضافه می‌کنیم، این برگ‌ها به‌عنوان Prefix مورد استفاده قرار می‌گیرند که بتواند اطلاعات Next Hop را در نزدیک‌ترین پدر خودشان که Prefix است را به ارث ببرد. حال Nodeهای میانی که به عنوان Prefix هستند را به Node معمولی تبدیل می‌کنیم. مزیت این کار این است که در حالت قبلی هر Nodeی ممکن است لازم باشد اطلاعات Next Hop را در خودش ذخیره کند. بنابراین برای همه Nodeها بایستی 1byte فضای Next Hop هم درنظر می‌گرفتیم.(دراسلاید قبلی دیدیم که اشاره‌گر سمت چپ و راست را خواهیم داشت و علاوه‌برآن فضایی هم برای ذخیره Next Hop نیاز داشتیم).
درصورتی‌که در حالت جدید Nodeهای میانی، اطلاعات Next Hop را در خودشان ذخیره نمی‌کنند و نیازی به این فیلد ندارند. اگر درخت‌مان بزرگتر شود، شما می‌توانید(طبق محاسبات) یک سوم از حجم حافظه برای Nodeهای میانی را حذف کنید. دلیلش این است که Nodeهای میانی فقط باید اطلاعات فرزند چپ و راست خودشان را نگه‌دارند و برای گره‌های برگ نیاز به اشاره‌گر به فرزندها نخواهیم داشت. بنابراین می‌توانیم فضای یکی از آن اشاره‌گر برگ را برای ذخیره کردن اطلاعات Prefix استفاده کنیم.

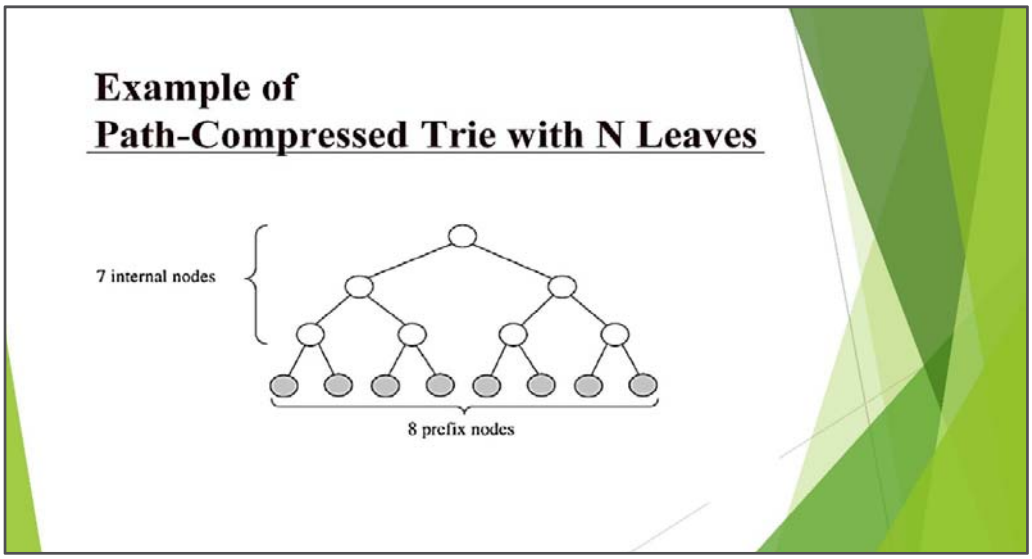
مشکل این روش: در این روش تعداد برگ‌های ما زیاد خواهدبود بنابراین تعداد Nodeهای ما بیشتر خواهدبود ولی Nodeها ساده‌تر شده و فضای کمتری را اشغال خواهد کرد. در مجموع به کمتر شدن فضای خالی حافظه‌مان می‌تواند کمک کند. پس برای تبدیل کردن درخت به Disjoint باید درخت را کامل کنیم. همه Nodeهایی که فرزند چپ یا راست ندارند را به آن‌ها باید فرزند چپ و راست بدهیم. باید گره‌ای به آن اضافه کنیم که بشود گره بزرگ، حالا مقدارش را چه بدهیم؟ از والد یا پدرخودش به ارث می‌برد. (مانند P1a در اسلاید).(از نزدیک‌ترین والد خودشان به ارث می‌برند.)



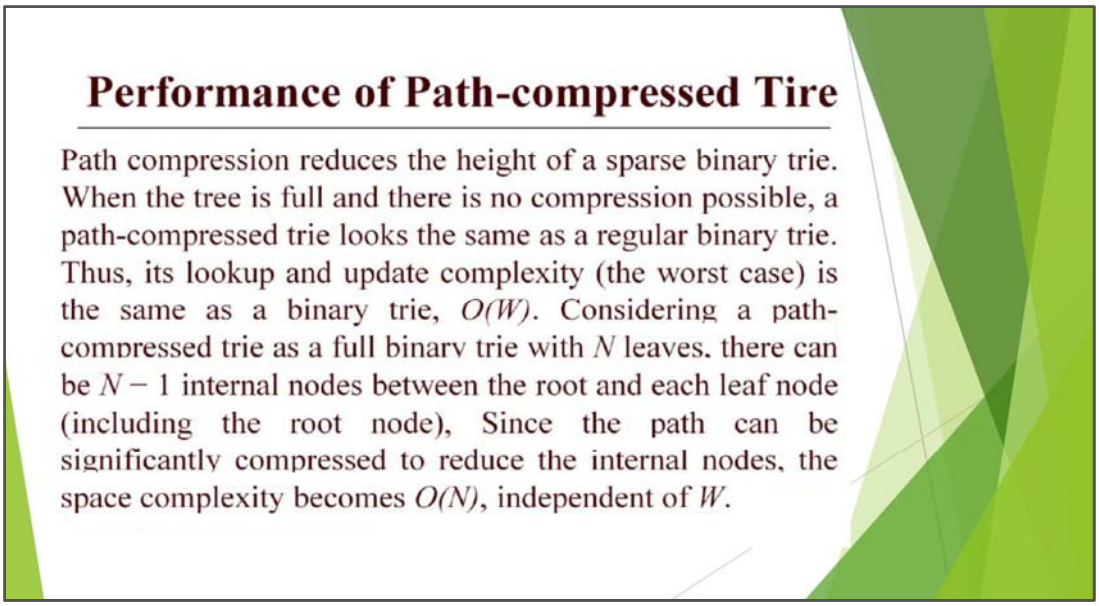
می‌توانیم درخت را کمی فشرده‌تر کنیم تا باز هم حجم را کاهش دهیم. از Path Compressed Tree استفاده می‌کنیم. فشرده‌سازی مسیرها در درخت اولیه، در برخی نقاط مسیرهایی‌ست که فقط به یک Prefix ختم می‌شوند. مثلا در درخت اولیه مسیر 00 را با هم چک کنیم: فقط به P3 ختم می‌شود بنابراین اگر این دو بیت در آدرس مقصد، 00 باشد را می‌توانیم پیمایش کنیم و اگر 01 باشد، از اول کار معلوم است که نمی‌توانیم پیمایش کنیم و بایستی شاخه دیگری پیدا کنیم. یا مثلا اگر در درخت اولیه از P6 تا F6 داریم، بنابراین یا ترتیب 011 را داریم که می‌توانیم پیمایش کنیم یا نمی‌توانیم. برای مواردی که فقط یک مسیر وجود دارد، می‌توانیم Nodeهای اضافی را حذف کنیم و به‌این ترتیب درخت‌مان را فشرده کنیم. از روی چند بیت پریده‌ایم و این بیت‌ها چه مقداری دارند برای ما مهم است. در ساختمان داده‌مان، دو فیلد جدید مطابق با اسلاید ۱ اضافه می‌کنیم. این دو فیلد برای درخت‌هایی که خلوت هستند و شاخه‌های طولانی دارند، می‌توانند به کم کردن حافظه کمک کند.

در اسلاید به P3 رسیدیم، Nodeی بوده که پریده‌ایم، Skip مساوی ۱ شده است یا برای P6 همینطور.

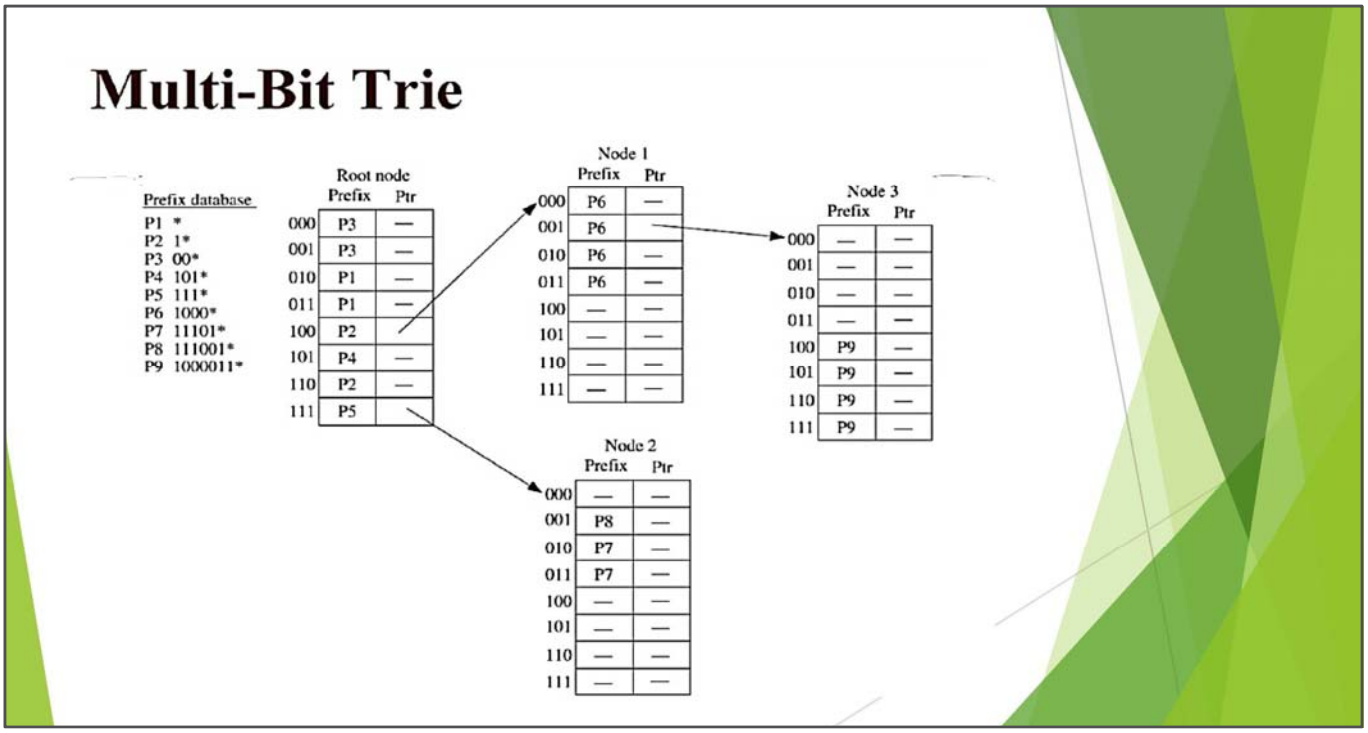
با اضافه کردن Skip Value نشان می‌دهیم چه اتفاقی رخ داده و چگونه درخت کوچک شده‌است.



نشان می‌دهد که اگر ۸ نود Prefix داشته باشیم، ۷نود میانی خواهیم داشت که می‌توانیم آن‌ها را فشرده‌تر کنیم اگر آن Prefix مقدار یکسانی داشته باشد.



کارایی روش Path Compress نشان داده شده‌است. برای درخت‌هایی که Sparse هستند و خلوت هستند، کارایی زیادی دارد ولی برای درخت‌های کامل کارایی مناسبی ندارد چون هیچ فشرده‌سازی ایجاد نمی‌کند پس کارایی این روش در بدترین حالت با روش باینری ساده‌ای که صحبت شد، هیچ تفاوتی نخواهد داشت. چون گره‌های میانی را فشرده می‌کنیم. O (N) برای مصرف حافظه می‌تواند باشد. (۷نود را می‌توانیم در مثال Compress کنیم).



به‌جای ۱بیت می‌توانیم از چند بیت استفاده کنیم. در درخت ۱بیتی می‌شود در هر بار فقط یک بیت را چک کرد. در این روش به جای ۱بیت می‌توان چند بیت را همزمان چک کرد. در این شکل یک حالت ۳بیتی را را ایجاد کرده. اصطلاحا با هین تعداد بیت‌هایی که چک می‌کند شاخه شاخه شدن Stride گفته می‌شود. اینجا ۳ است.

با ۳بیت ۸=$۲^۳$ حالت می‌توان ساخت.

۸حالت را در اسلاید برای ما مشخص کرده است. هر Node آن Tree که شامل ۸عدد Entry است را ایجاد می‌کنیم Prefixهایی که طول کمتر از Stride را دارند. به‌عنوان مثال همه این Prefixها طولشان از ۳ بیشتر نیست. (P1,P2,P3) به حالت ۳بیتی باید آن‌را توسعه دهیم. مثال: اگر P2 را در نظر بگیریم، اولش ۱ است، بعد دو بیت بعدی ۴حالت را می‌تواند داشته باشد، باید به سه بیت تبدیل شود.می‌شود: 111 110 101 100 تا سه بیت شود.

حالت‌هایی که با Prefix طولانی‌تری Match شود، مثلا با P4 و P5 آن‌ها را به‌همان اسم می‌گذاریم ولی مابقی را به P2 تبدیل می‌کنیم. سه صفر بیشترین Match را با P3 دارند. آن‌را می‌نویسیم. یا

100بیشترین تطبیق را با P2 دارد چون فرمان دوصفر اضافه کرده‌ایم. وقتی این‌ها را پر کردیم، حالت‌هایی که مشترک هستند مثل 100، هم P2 هم P6 و هم P9 را نشان می‌دهد.

100 می‌تواند P2، P6 و P9 باشد. با استفاده از Nodeهای بعدی مشخص می‌کنیم کدام‌یک از این Prefixها مدنظر ماست. مثلا اگر 100 بود ⟵ P2

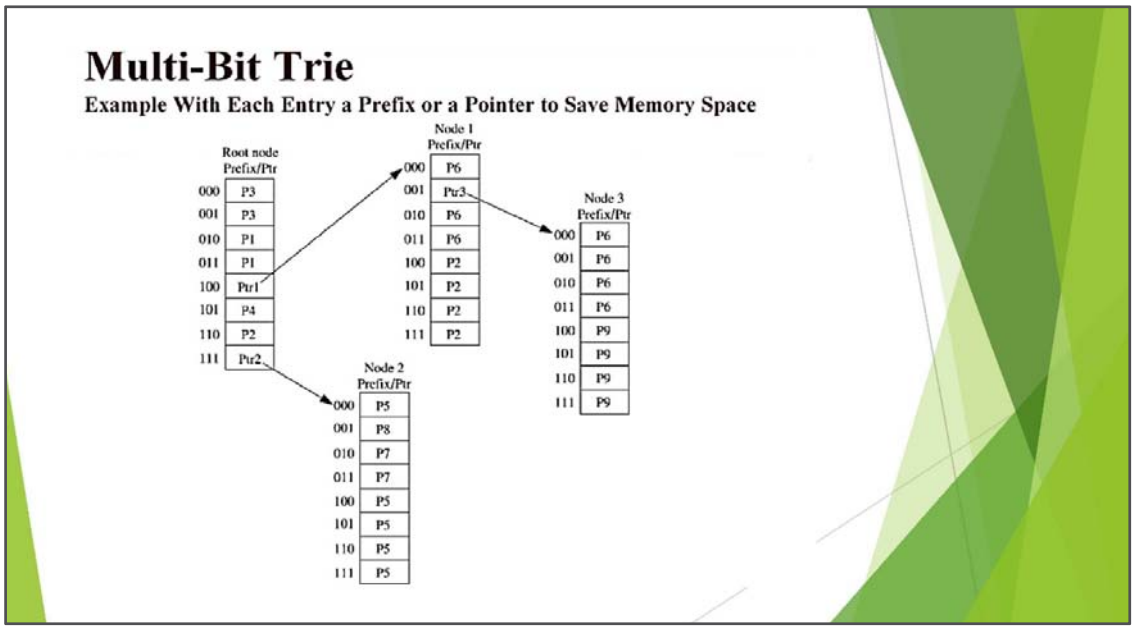
اگر 000 شد ⟵ (Longest Match) می‌شود حالت P6

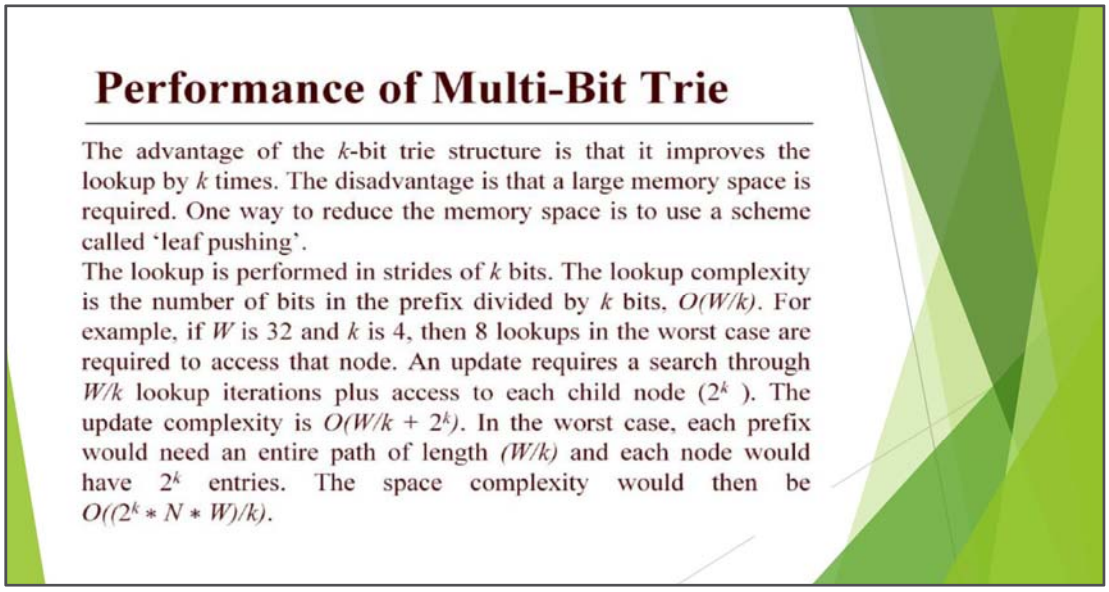
در درخت ایجاد شده در این اسلاید هر سه بیت را در یک گام می‌توانیم بررسی کنیم. چون هر Node درخت را یکبار از حافظه می‌خوانیم، اگر اندازه‌ی ماکزیمم Prefixها برابر با Wبیت باشد و Strideها K باشد، تعداد دفعات مراجعه به حافظه حداکثر برابر W/K خواهدبود. در هر بار می‌توانیم Kبیت را بررسی می‌کنیم. پس پیچیدگی جستجو می‌شود. O (W/K)

سرعت می‌تواند نسبت به حالت تک بیتی Kبرابر باشد.

مشکل این است که ما مجبوریم یکسری Prefixها را تکرار کنیم، مثلا دیدیم که P2 و P6 را چندین مرتبه تکرار کردیم دلیلش این است که می‌خواهیم Prefixها را مضربی را از K بکنیم که ما را دچار مشکل می‌کند. K هر چه شود براساس آن همه Prefixها آن‌را شامل می‌شود. این‌کار حافظه مصرفی ما را افزایش می‌دهد. در بدترین حالت حجم می‌تواند به 2^kافزایش یابد.

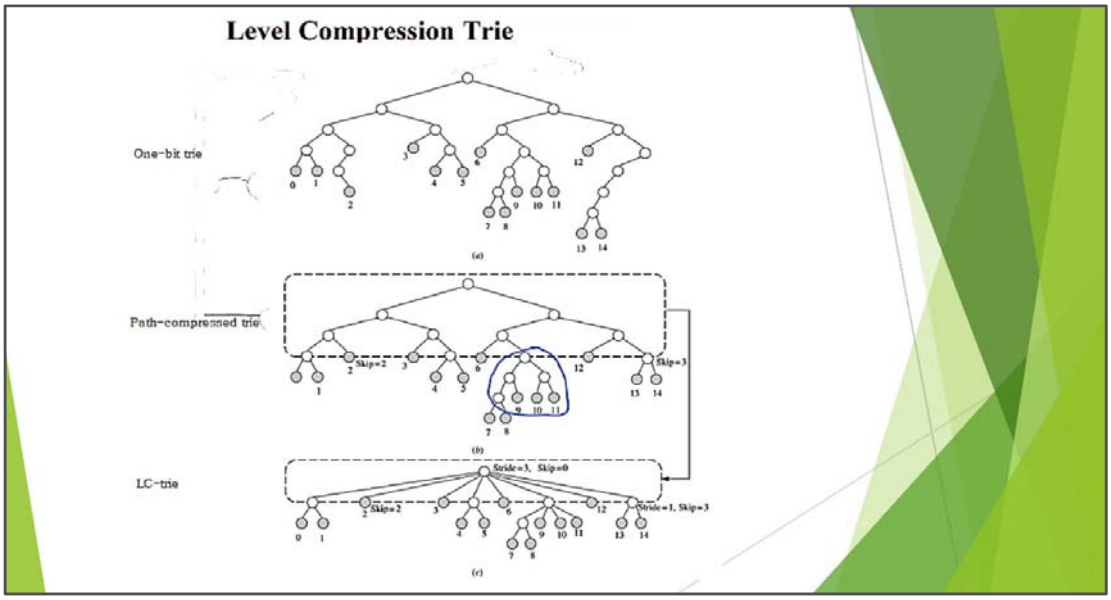
اگر مثل Disjoint، Prefixها را در K، Push کنیم، درخت ما تبدیل به چنین حالتی خواهد شد، حالت Pointerها را نیاز نخواهیم داشت. P2 نوشته نمی‌شود، Pointer می‌کنیم به خود شماره۱. براساس سه Nodeبعدی تصمیم می‌گیریم اگر 000 بود P6 می‌شود، اگر سه Node بعدی 001 بود مجددا Pointer دارد و همینطور تا آخر. (حالت قبلی گفتیم ۱اضافه می‌کنیم. 10 و 11 مشخص کننده این است که کدام حالت را پیش روی خواهیم داشت)



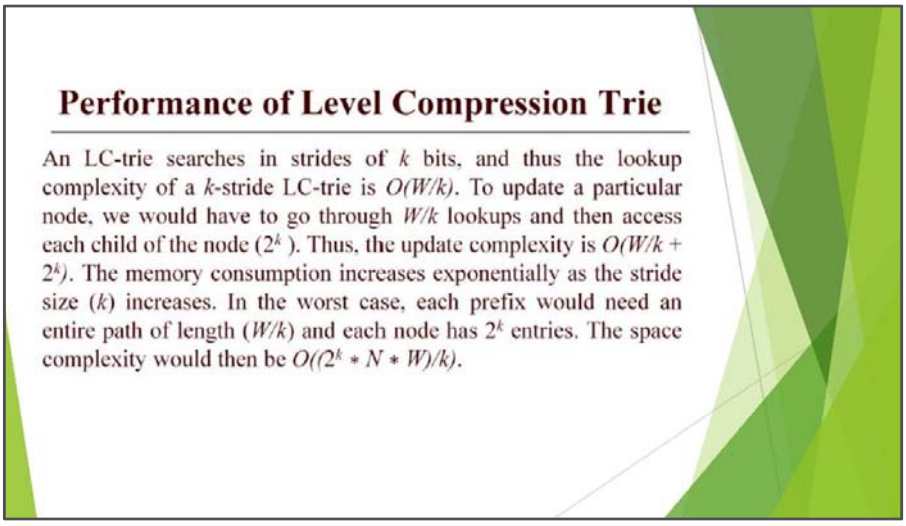


با حذف کردن Pointerها حجم حافظه را نصف کردیم، پیچیدگی این روش $O(W/K+2^k)$.

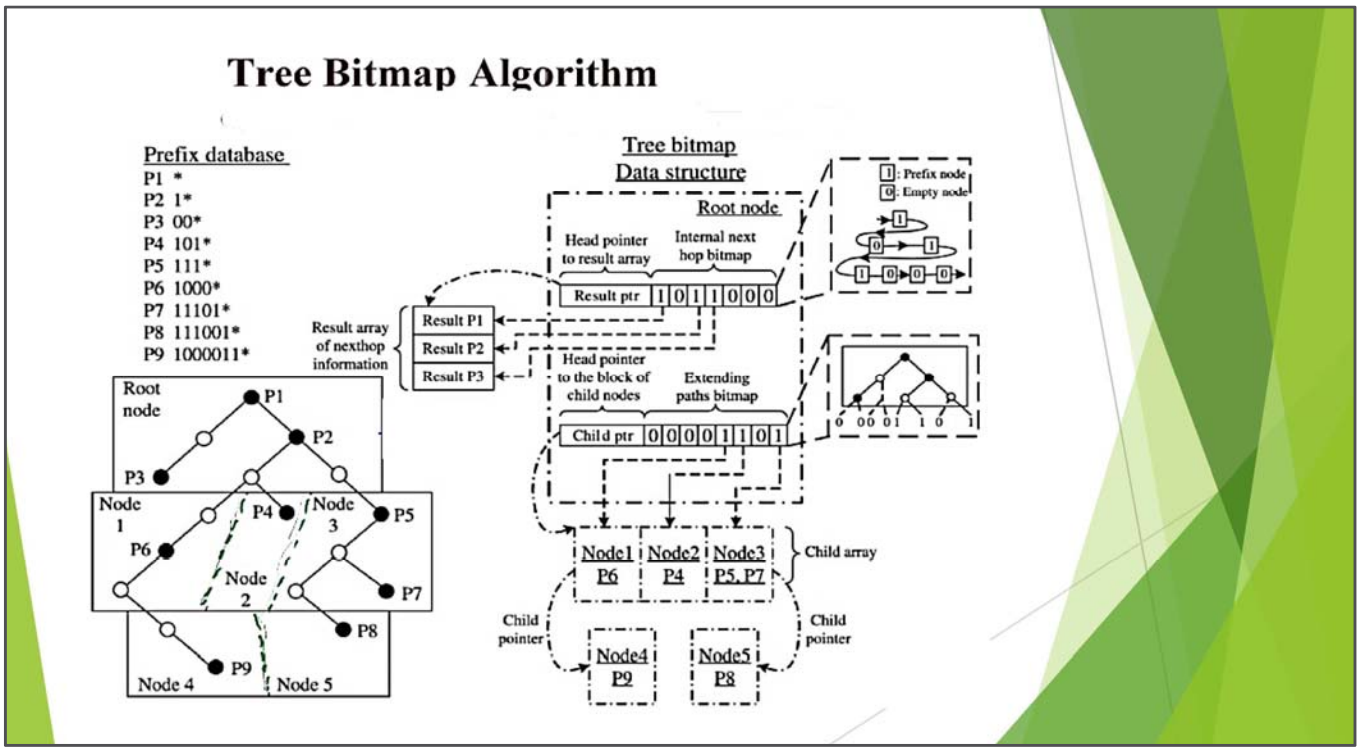
kای که انتخاب می‌شود برای ما مهم است. پیچیدگی جستجو W/K خواهد بود. بروزرسانی هم همینطور و 2^k برای بدترین حالت باید بررسی شود. برای همین در بدترین حالت، پیچیدگی $W/K+2^k$ خواهد شد. در انتخاب K بایستی Trade Of ایجاد کنیم، اگر K کوچک باشد، تعداد مراجعه به حافظه زیاد می‌شود ولی حجم حافظه کم. چون سه بیت راچک می‌کنیم و اگر K بزرگ شود تعداد دفعات مراجعه شده ولی حجم حافظه بیشتر می‌شود.



روشی داریم به نام Level Compress Tree که بر اساس 1bit tree و Path Compression یک Level Compression هم اضافه کرده است. در این روش، روش چند بیتی و فشرده‌سازی را به‌صورت همزمان استفاده کرده می‌کند . هرجایی که شاخه Sparseدارد شاخه‌هایش پر نیست، فشرده‌سازی کرده و هر جا که شاخه‌ها کامل هستند به‌صورت چند بیتی کار می‌کند.مطابق با اسلاید ۳ سطح اول را که کامل هستند به روش ۳بیتی عمل می‌کند و تعداد زیادی از Nodeها می‌توانیم حذف کنیم. در سطح بعدی شاخه ها را به روش فشرده‌سازی انجام داده. برای شاخه‌هایی که تنها هستند. (در انتها ۳ Node را Skip کرده تا به اینجا رسیده است). Skip: 1 (Stride)K=1



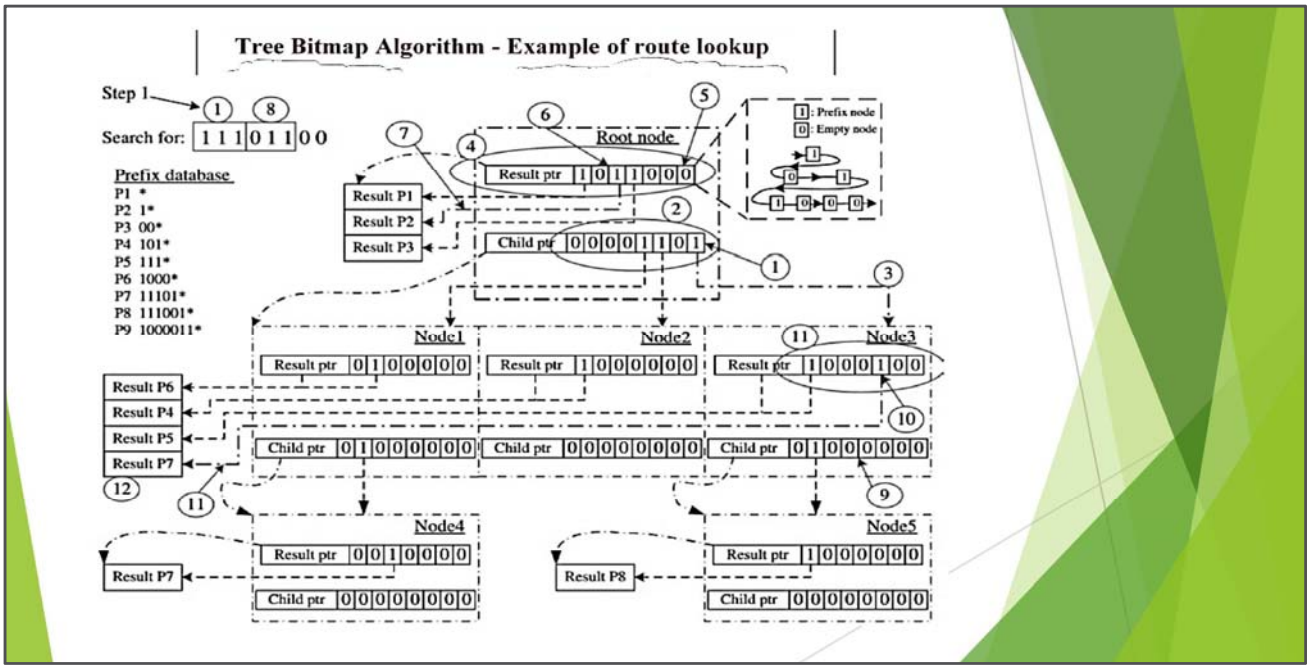
فشرده‌سازی در بدترین حالت شبیه حالت چند بیتی می‌شود. اشکالی که وجود دارد این است که ساختار پیچیده‌تر می‌شود چون دو روش را داریم با هم ترکیب می‌کنیم. همان تعاریف اینجا هم پا برجا هستند.



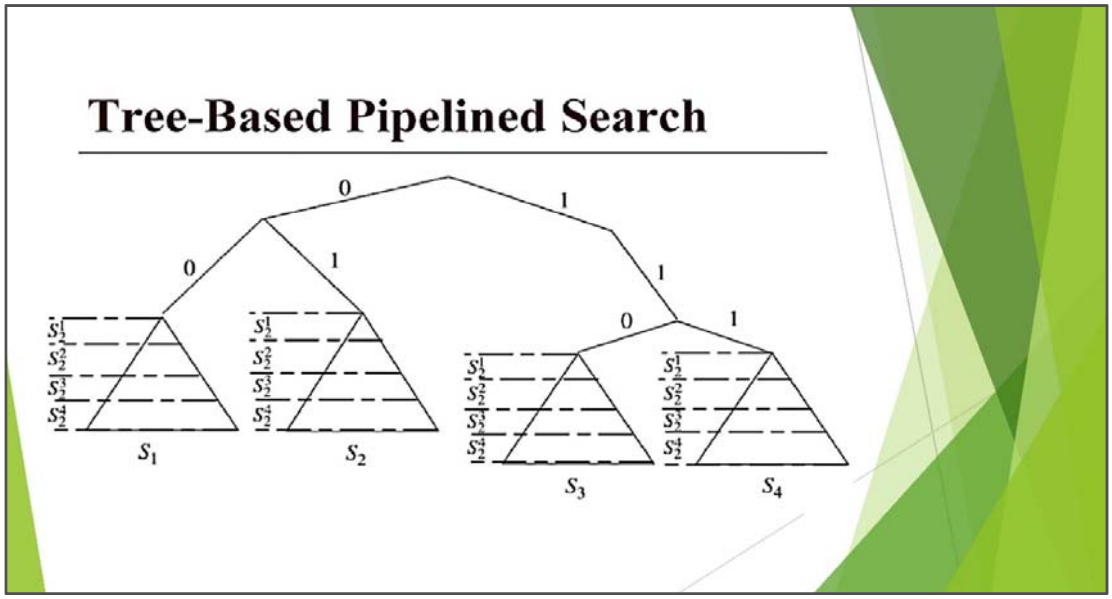
روش Tree Bitmap: از روی درخت باینری یک Bit Mapی تهیه می‌کنیم. بجای این‌که درخت را نگهداری کنیم Bit Map را نگهداری می‌کنیم. به‌عنوان مثال اگر St Ride برابر ۳ بگیریم، سه سطح اول از ریشه به‌عنوان یک Node از درخت جدا می‌شود. دوباره از گره بعدی سه سطح جدا می‌کنیم. برای گروه‌های بعدی هم همین اتفاق می‌افتد. اگر بخواهیم سه بیت را چک کنیم، سطح اول می‌شود یک Node، سه‌تای بعدی می‌شود یک Node و سه تای بعدی می‌شود Node بعدی. با این روش درخت به یک Node ریشه می‌آید و به پنج Node دیگر تقسیم می‌شود. یک گره ریشه دارد و پنج Node را شامل خواهد بود. برای هر Node دو عدد Bit Map نگهداری می‌کنیم. مثل مثال سمت چپ اسلاید.(Result & Child Pointer)را نشان می‌دهد.

دو بیت مپ نگهداری می‌کنیم یعنی:در درخت باینری چه Nodeهایی Prefix جدول هستند و چه Nodeهایی Prefix نیستند. P1, P2, P3 نیستند. شماره‌گذاری‌شان هم به این شکل انجام دهیم که هر سوپر Node درخت، حداکثرمی‌توانیم ۷گره از درخت اصلی را داشته باشیم. این درخت را کامل کنیم.(در هر سطح در Node ریشه می‌توانیم ۷Node راداشته باشیم)

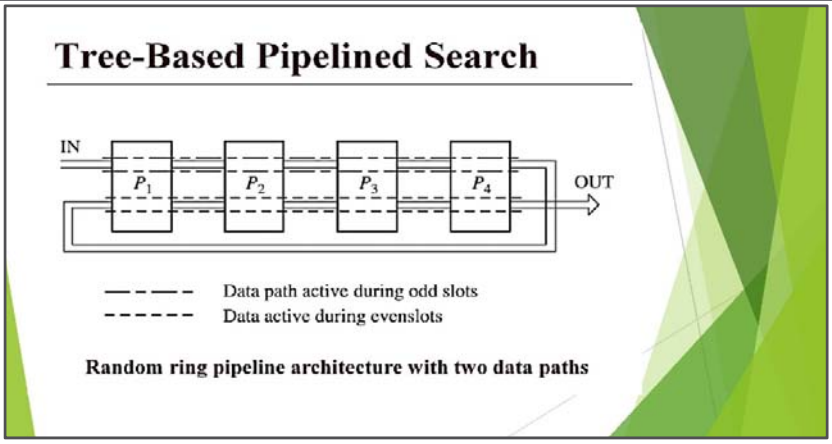
فرض کنید همه ۷گره وجود داشته باشند، برای Prefixها عدد۱ قرار می‌دهیم، برای گره‌های معمولی و خانه‌های خالی صفر را قرار می‌دهیم.(همان اتفاقی که در شکل نشان داده شد). (دقیقه ۴۰ کپچر)



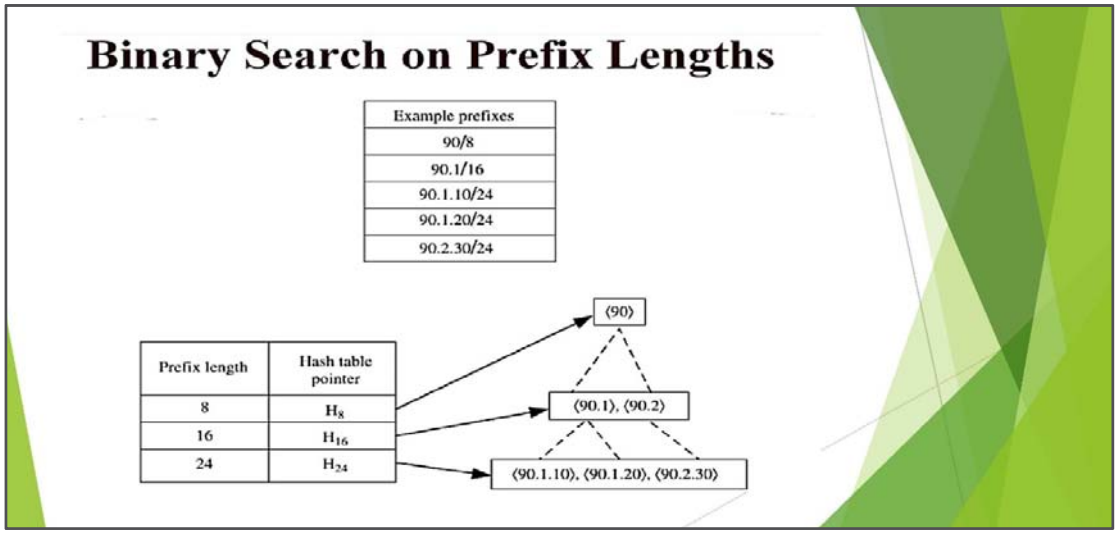
می‌توانیم ساختار قبلی را به‌صورت کاهش حجم و سادگی به‌این‌صورت نمایش داد. هر ۷گره معمولی یک سوپر نود ایجاد می‌کنیم که فضای ۲بیت Map ۷بیتی و ۸بیتی را خواهد داشت، به این‌صورت دو Pointer هم به آن اضافه خواهد شد.



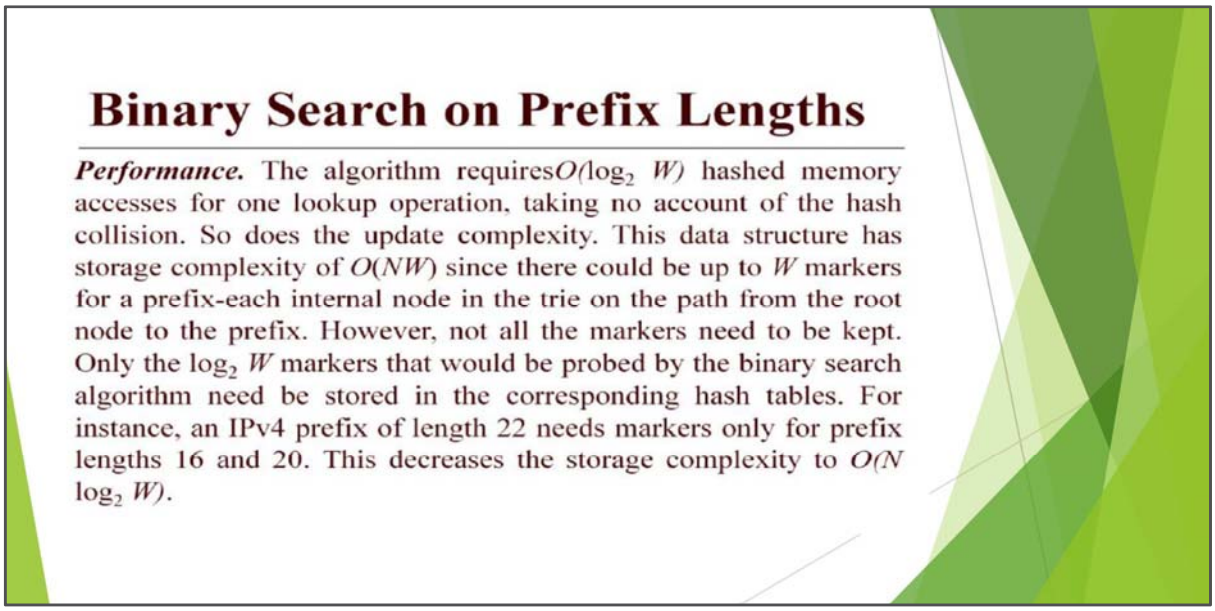
وقتی حافظه‌ای داریم و روی آن عمل Look Up انجام دهیم در هر لحظه، فقط می‌توانیم یک جستجو انجام دهیم. ولی اگر حافظه یک درخت را به چندین بخش اختصاص دهیم و کپی کنیم، می‌توانیم از تکنیکPipelined استفاده کرده و جستجوی متفاوتی انجام دهیم.



یک درخت جستجو را به ۴قسمت تقسیم کرده، یک بخشی از جستجو در حافظه اول انجام می‌شود، نتیجه در بخش دیگر. حال می‌توانیم به‌صورت همزمان یک جستجوی جدیدی در بخش اول انجام دهیم به‌همین صورت می‌توانیم چهار جستجو را به‌صورت Pipeline انجام دهیم و سرعت را ۴برابر کنیم.فرض کنید دنبال Prefixای هستیم. ۱،۲،۳،۴ منظور این است که اگر بتوانیم حافظه را به ۴بخش تقسیم کنیم می‌توانیم به‌صورت متوالی این عملیات را انجام دهیم به‌شرط این‌که این سطوح مختلف درخت را بتوانیم انجام دهیم. جستجو در یک سطح تمام شد. ادامه جستجو در سطح بعدی امکان پذیر است.

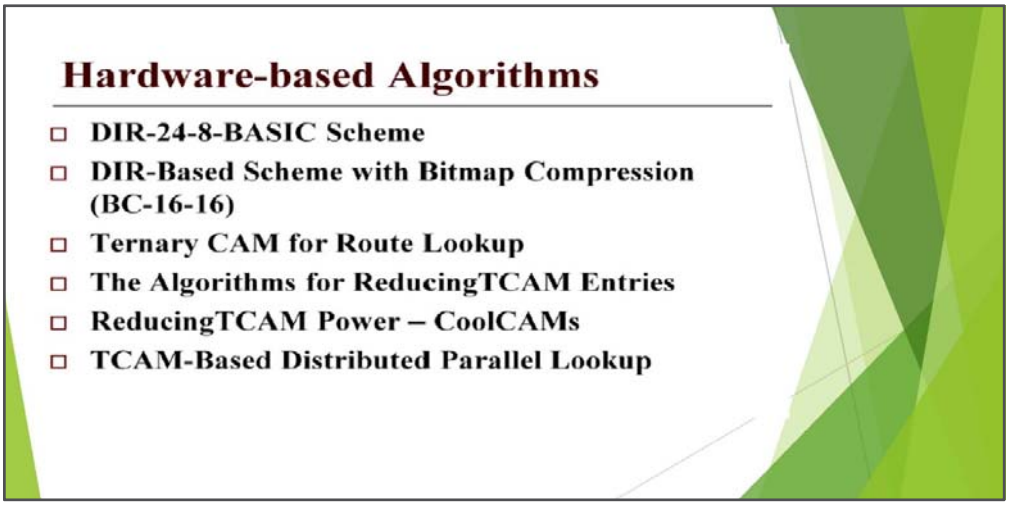


مثالی نمایش یافته‌است. سطح‌بندی براساس طول Prefix انجام شده‌است. طول ۸:۱۶:۲۴ و یک Pointer هم به سه تا جدول ایجاد کرده‌است. اگر Prefix، ۸باشد، ۱عدد Entry بیشتر نداریم. در حافظه آمده‌است که به‌صورت مستقیم آن را سرچ می‌کند و دیگر نیازی نیست دوتای بعدی را سرچ کند. اگر ۱۶ است. 90.1 و 90.2 را قرار داده اگر ۲۴ باشد ۱۰۲۰۳۰ را مشخص کرده به‌این‌صورت براساس Prefix معلوم می‌شود کدام جدول بررسی شود. یک‌سری Pointer هم اضافه شده. مثلا اگر دنبال 90.2. 30. 20 :IP هستیم به‌صورت سلسله مراتب چک می‌کند. 90 چک می‌شود← 90.2 و بعد 90.2. 30 می‌شود Longest Match و ادامه کار.



می‌دانیم که روش Binary Search، سریعترین روش جستجو است. (نصف کردن داده‌ها به‌شرط Sort بودن، پیدا کردن میانه، حذف یک‌طرف و...) بنابراین O(Log(n)) بود پیچیدگی الگوریتم. در این روش یک شیبه‌سازی جستجوی دودویی مدنظرمان است. در جستجوی دودویی باید اطلاعات به‌صورت Sort شده‌باشند. حال زمانی‌که می‌خواهیم عددی پیدا کنیم فضای جستجو را به دو قسمت تقسیم می‌کنیم. برای P1.3 جدول Range و Prefix را تشکیل داده. Rangeهایی را درست کرده و در اشکال B و C نشان داده‌است.

از حالت 100000 شروع کرده و این Rangeبندی‌ها را ادامه داده در هر بار اندازه فضای Search نصف می‌گردد. از این ایده در Routing Table می‌توانیم استفاده کنیم. اشکالش این است که اطلاعات جدول عدد نیستند، بلکه Rangeی را برای ما مشخص می‌کنند. می‌توانیم شروع و پایان رنج‌ها را مشخص کنیم. هر IP یک رنج برای ما مشخص می‌کند. آدرس مقصد که در یکی از این رنج‌های مرتب شده قرار گرفت، بسته متعلق به آن زیر شبکه خواهد بود. اگر حالت‌های منطقی برای رنج‌هایش بیابد مثلا بزرگتر مساوی بایستی متناسب آن پورت را تعیین کنیم. رنج‌های شروع و پایان را که می‌نویسیم در اصل داریم اندازه جدول مسیریابی را دو برابر می‌کنیم. بعلاوه یکسری اطلاعات را هم به Entryها اضافه می‌کنیم. در این روش داریم اندازه حافظه را بزرگتر می‌کنیم اما Searchها سریعتر خواهدشد.



روش‌های سخت‌افزاری جستجوی اطلاعات مسیریابی:

روش‌های سخت‌افزاری، سرعت بسیار بالاتری دارند. اما هزینه پیاده‌سازی بالاست و انعطاف‌پذیری(Flexibility) کمتر است.

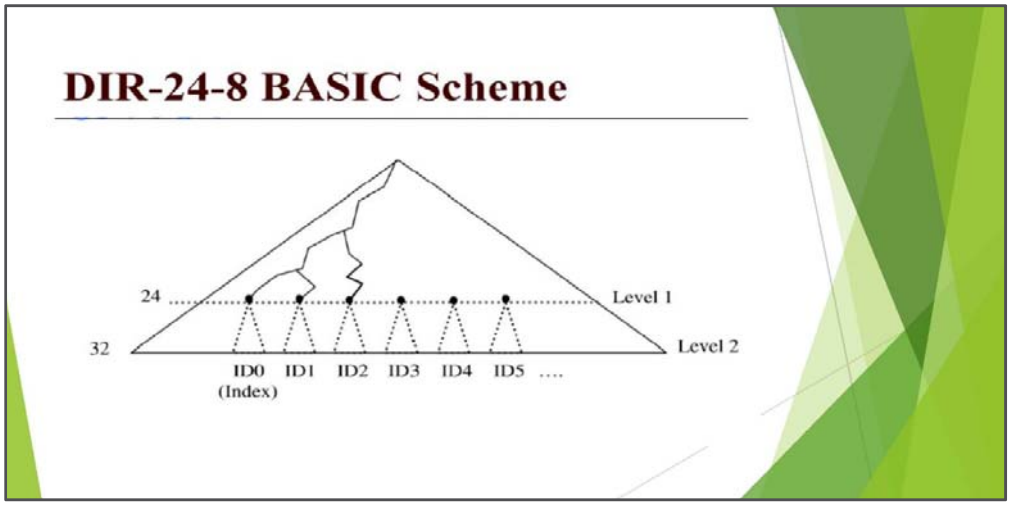
به‌طور کلی روش‌هاس سخت‌افزاری دو دسته‌اند:

۱. DIR
۲. Ternary

روش DIR (Direct): مستقیم: با یکبار مراجعه به حافظه، جستجو را انجام می‌دهد. حجم حافظه مصرفی بالاست. هزینه Update کردن جداول بالاست.

روش Ternary: مبتنی بر Content Addressable Memory است.

این‌ها براساس محتوا، آدرس‌دهی را انجام می‌دهند.

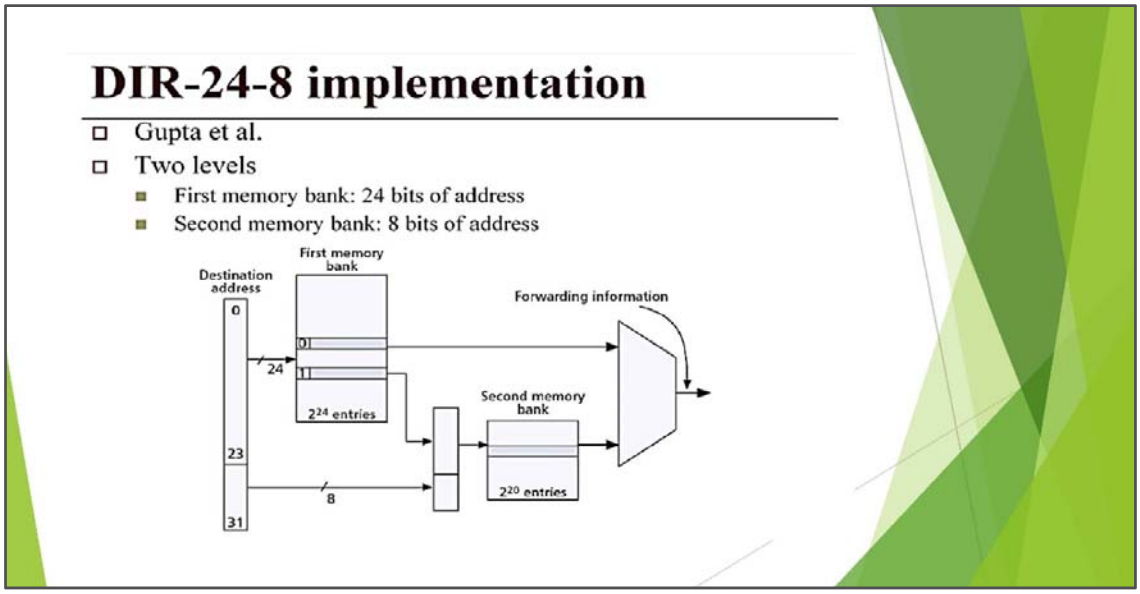


روش‌های Direct دو شماره دارند. مثلا DIR-24-8 یعنی یک جستجوی دو سطحی خواهیم داشت درخت جستجو، درختی با عمق ۳۲ است. اگر همه برگ‌ها را در یک حافظه بخواهیم ذخیره کنیم، نیازمند 2^{32}=4GB هستیم. می‌توانیم آدرس مقصد بسته را به‌عنوان آدرس حافظه استفاده کنیم و به خانه حافظه اشاره کنیم در محتوای حافظه هم Next Hop را ذخیره کنیم. به‌این ترتیب با یکبار مراجعه به حافظه می‌توان عمل Look Up را انجام داد. اما از نظر هزینه پیاده‌سازی، حدود 4GB حافظه نیاز خواهیم داشت.

اگر نیاز به Update جداول باشد، هزینه بیشتر خواهد شد چون هر تغییر در جدول مسیریابی، می‌توان سبب هزاران تغییر درجه اول Direct باشد. (چون Pointerها هم مورد اشاره‌اند).

طبق اطلاعات آماری که از جداول مسیریابی به‌دست آمده‌است مشاهده شده که اکثر Entryها Prefix کمتر از ۲۴ دارند و تعداد کمی از Prefixها طولشان از ۲۴ بیشتر است. به‌همین دلیل دو سطح ۲۴ و ۸ تعریف شده‌است. پس یک راه‌حل این است که سطح ۲۴ اطلاعاتش ذخیره شود. تعداد Nodeها در این حالت 2^{24} یعنی ۱۶مگابایت که در برابر ۴گیگ بسیار کمتر است. حال اگر Prefix کمتر از طول ۲۴ داشته باشیم، در عمق حداکثر ۲۴ می‌شود Next Hop را پیدا کرد. اگر Prefixهایی با طول بیشتر از ۲۴ داشته باشیم، بایستی عمل جستجو را بیش از ۲۴، ادامه دهیم و تا عمق ۳۲ این امکان است که این اتفاق بیفتد.

برای Prefixهایی که طول بزرگتر از ۲۴ دارند، یک حافظه جداگانه درنظر بگیریم. از آن‌هایی که تعداد این‌ها کم است، نیازی به حافظه بزرگی نداریم پس ساختار و معماری‌مان شبیه شکل اسلاید ۴ می‌شود.

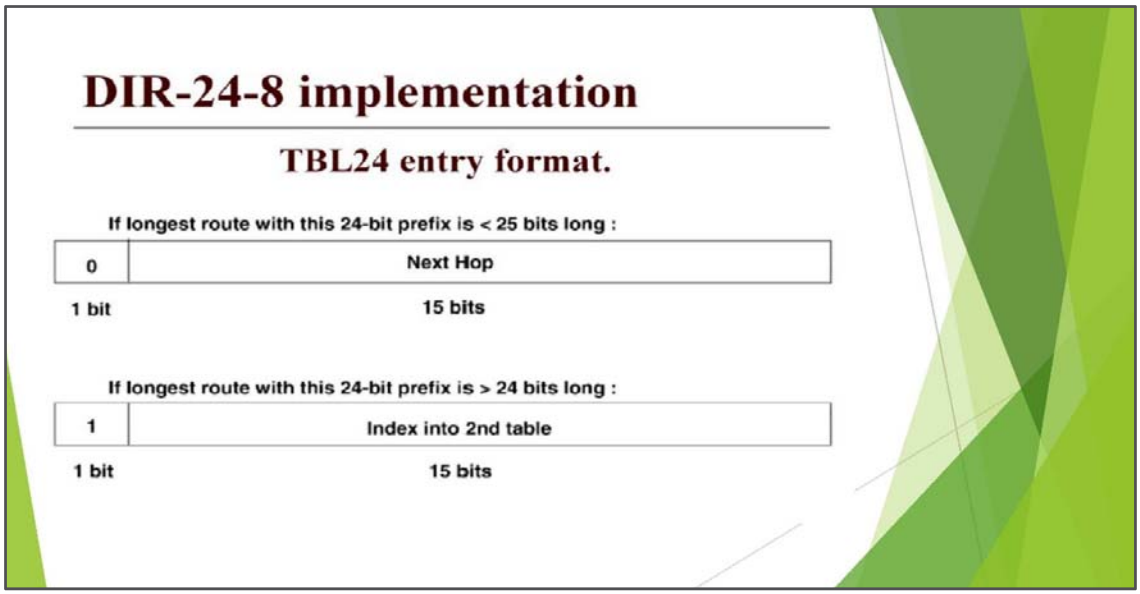


پیاده‌سازی سخت‌افزاری یک Tree چند سطحی نشان داده شده‌است. دو سطح دارد: یک سطح همان ریشه می‌توان فرض کرد با۲۴بیت و یک سطح ۸بیت باقی‌مانده را تصمیم‌گیری می‌کند.

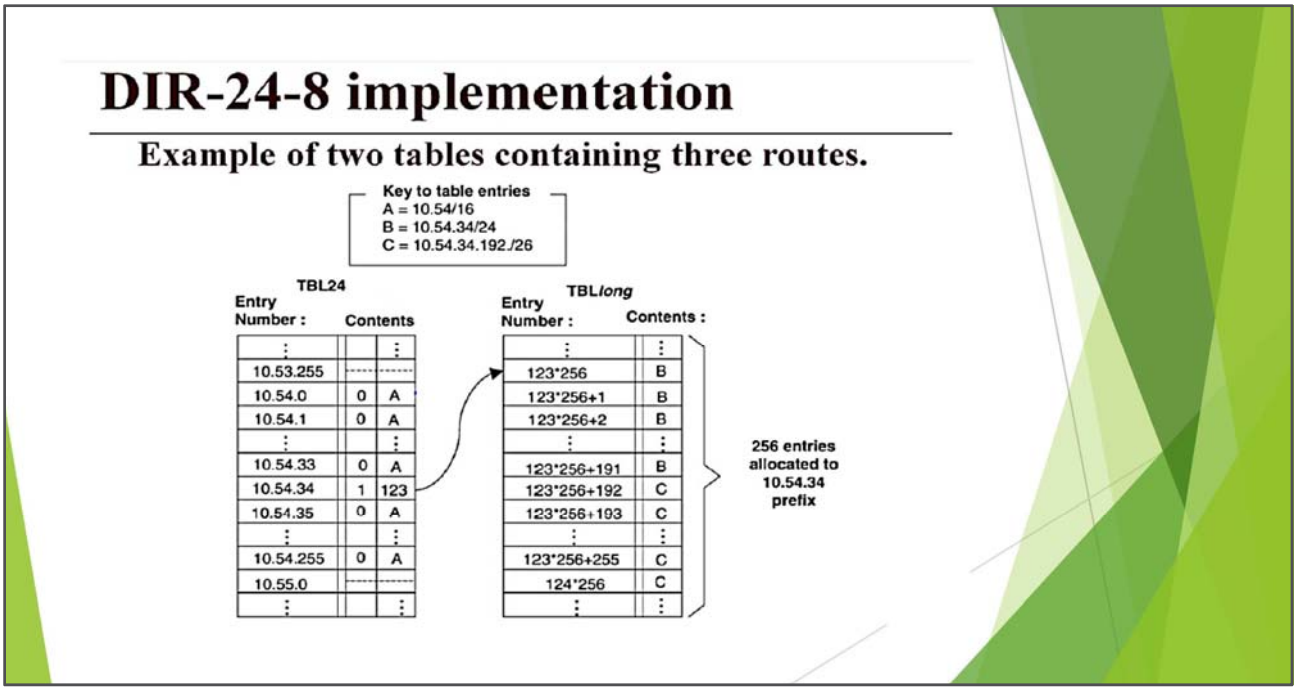
یک بانک حافظه با $۲^{۲۴}$ Entry را در نود ریشه پیاده‌سازی کرده و براساس ۱۴بیت اول به حافظه مراجعه می‌کند با یافتن طولانی‌ترین تطبیق(Prefix (Longest Match مورد جستجو را پیدا می‌کند یا به اشاره‌گر نود بعدی، (بانک دوم) اشاره می‌کند و به آن ارجاع می‌دهد. ۸بیت باقی‌مانده سراغ بانک دوم می‌رود، آنجا هم Longest Match را بایستی انجام دهیم تا مشخص شود که چه Prefixی مناسب است. برای این‌که Throughputمان بهتر شود می‌توان دو بانک حافظه‌ای را به‌صورت Pipe(لوله‌ای) کنار هم قرار داد و برای استفاده از حافظه‌های پویا که زمان دسترسی می‌تواند 50ns باشد، می‌توان در حدود ۲۰میلیون جستجو در ثانیه انجام دهد.

مشکل پیاده‌سازی این روش این است که برای بسط دادن آن ممکن است به تعداد زیادی خانه حافظه، نیاز پیدا کنیم و مجبور شویم خانه‌های حافظه را زیاد تغییر دهیم. ضمنا به‌علت حجم حافظه بالای مورد نیاز، امکان پیاده‌سازی آن در حافظه‌های ایستا(که از پویا سریع‌تر است) وجود ندارد. می‌توانی بانک‌ها را سه سطحی کنیم تا حجم حافظه هر بانک، کمتر شود. هرچند به بدترین حالت جستجو یک دسترسی به حافظه افزوده خواهد شد. به‌جای دوبار، سه‌بار مراجعه به حافظه. آن‌وقت خواهیم‌داشت: DIR-21-3-8

۲۱بیت اول بعد ۳بیت و درنهایت ۸بیت.



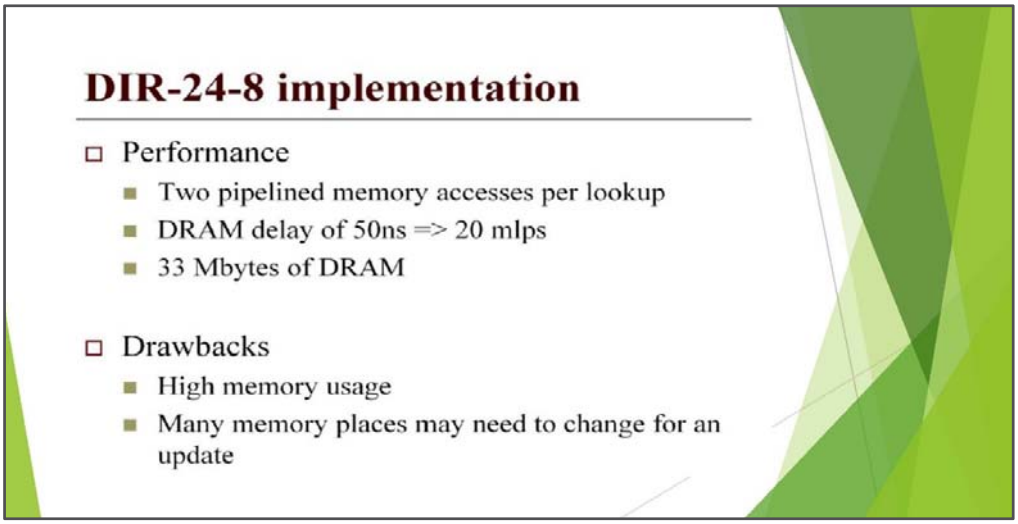
در این مثال محتوای حافظه سطح اول را نشان می‌دهد که اگر بیت اول صفر باشد یعنی Next Hop در آن ذخیره شده و اگر بیت اول ۱ باشد محتوای آن Indexای است که به جدول دوم اشاره می‌کند. می‌توان برای خانه‌های حافظه از آن استفاده کرد.



همان مسئله تکرار شده‌است اگر خانه اول صفر باشد یعنی A، Next Hopهاست ولی اگر ۱ است مشخص کننده آدرس یا Indexی در جدول حافظه‌هاست. حال در این مثال فرض کرده ۳عدد Entry در جدول‌ها باشد، اول باید رکوردهایی که طول کوچکتر از ۲۴بیت دارند را ۲۴بیتی کنیم. مثلا 10.54 باید به 10.54.0 تا 10.52.255 یعنی به آن Range دهیم چون کمتر از ۲۴ است و ما داریم درباره DIR248 حرف می‌زنیم باید حداقل به ۲۴ افزایش دهیم. در همه حالات به‌غیر از حالات B و C، Next Hopها A خواهد بود. برای حالت B و C صرفا با استفاده از 10.54.34 نمی‌توانیم تصمیم بگیریم Next Hop چیست. بایستی آدرسی به Index حافظه سطح دوم ایجاد کنیم و با بیت‌های باقی‌مانده(۸بیت) تصمیم بگیریم، Next Hop چه خواهد شد. با ۸بیت باقی‌مانده شروع کرده و تا ۲۵۶ جلو رفته‌است. تمامی حالات را در حافظه سطح دوم گنجانده است.

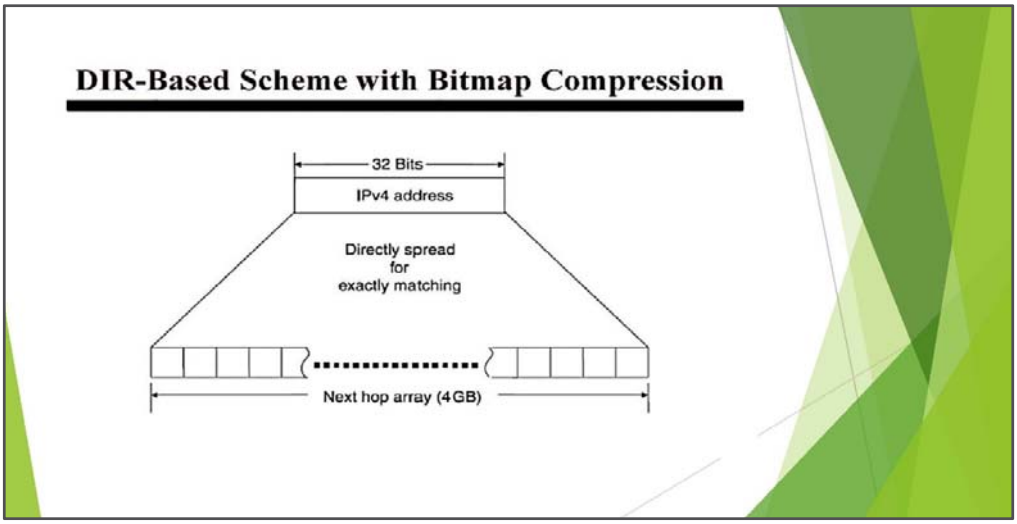
بسته‌ای گرفتیم با آدرس 10.54.34.185 ابتدا ۲۴بیت اول را چک می‌کند حافظه سطح اول Indexی به ما می‌دهدکه باید سراغ جدول سطح دو برویم.← سراغ ۸بیت باقی‌مانده ← ۸بیت باقی‌مانده 185 را نشان می‌دهد. 185 مشخص کننده B خواهدبود. می‌بینیم که یک Entry داخل جدول روی تعداد زیادی از خانه‌ها در سطح اول و دوم تکرار می‌شود و این ایرادی است که در این روش وجود داردو باعث افزایش حجم حافظه می‌شود.

سوال:چرا بعد از ۲۵۶ هنوز ادامه دارد؟ چون داریم درباره Longest Match صحبت می‌کنیم. بعد از آن هرچه باشد، Longest Match می‌شود به‌همین دلیل تا C ادامه دارد. همه حالت‌ها پورت A هستند. فقط 10.54.34 مشکل ساز خواهد شد که برای آن Index تعریف کرده تا بتواند آن‌را در جدول مشخص کند. Index را ۱۲۳ درنظر گرفته‌است. 156.1.2 تا برسد به ←۱۹۱ تا ۱۹۱، B پورت خروجی خواهد بود و از آن‌جا به بعد پورت C برای خروجی درنظر گرفته می‌شود.

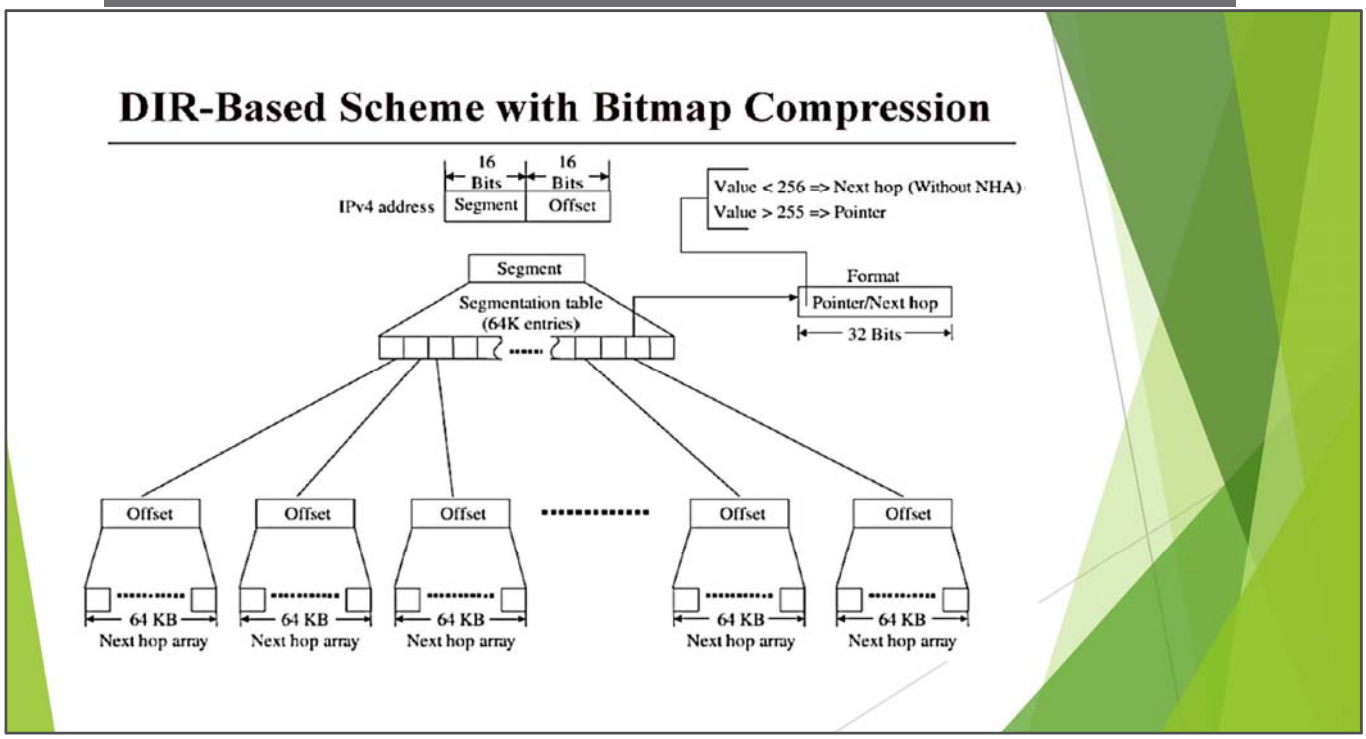


در این روش جستجو با یک یا دوبار به حافظه انجام می‌شود. ولی اگر بخواهیم تغییراتی را در جدول بدهیم مجبور خواهیم بود تعداد زیادی از خانه‌های حافظه را تغییر دهیم.

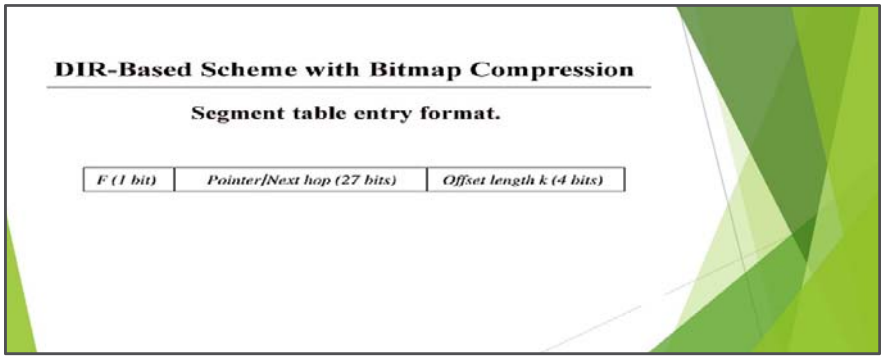
اشکال اصلی: حافظه مصرفی زیاد و تغییرات زیادی برای Update نیاز است.



روش دیگری داریم به‌نام ذکر شده در اسلاید که به آن DIR-16-16 هم گفته می‌شود.



از ایده Direct دو سطحی در آن استفاده شده‌است. اما سعی شده اندازه عمق حافظه سطح۲ دیگر ثابت نباشد. در این روش تا ۱۶ ادامه می‌دهد. بعد درخت شکسته می‌شود و همه Prefixها را با ۱۶ مشخص می‌کنیم. بعضی از Prefixها که طول بزرگتر از ۱۶بیت داشته باشند، عمق بزرگترین شاخه را درنظر می‌گیریم، حداکثر تا ۳۲ می‌تواند برسد. (کمتر بود تا همان‌جا را نگه‌می‌داریم) در حافظه دوم به‌جای این‌که به‌ازای هر شاخه $۲^{۱۶}$تا حافظه را در سطح۲ درنظر بگیریم، به‌اندازه‌ی Max عمق درنظر می‌گیریم. (عمق ثابت نیست). عمق Kبیت باشد، 2^k خانه حافظه در نظر گرفته می‌شود. به‌این ترتیب در حافظه سطح اول یا Next Hop پیدا خواهد شد یا Pointer به حافظه سطح دوم خواهیم داشت. اگر Pointer به حافظه دوم داشته باشیم یک حافظه ۴بیتی را درنظر می‌گیریم که عمق سطح دوم‌مان را برایمان مشخص کند.



یک بیت مشخص شده اگر Next Hop است، که هیچ، اگر Pointer است Offsetی که در ۴بیت بعدی است مشخص کننده خواهد بود.