

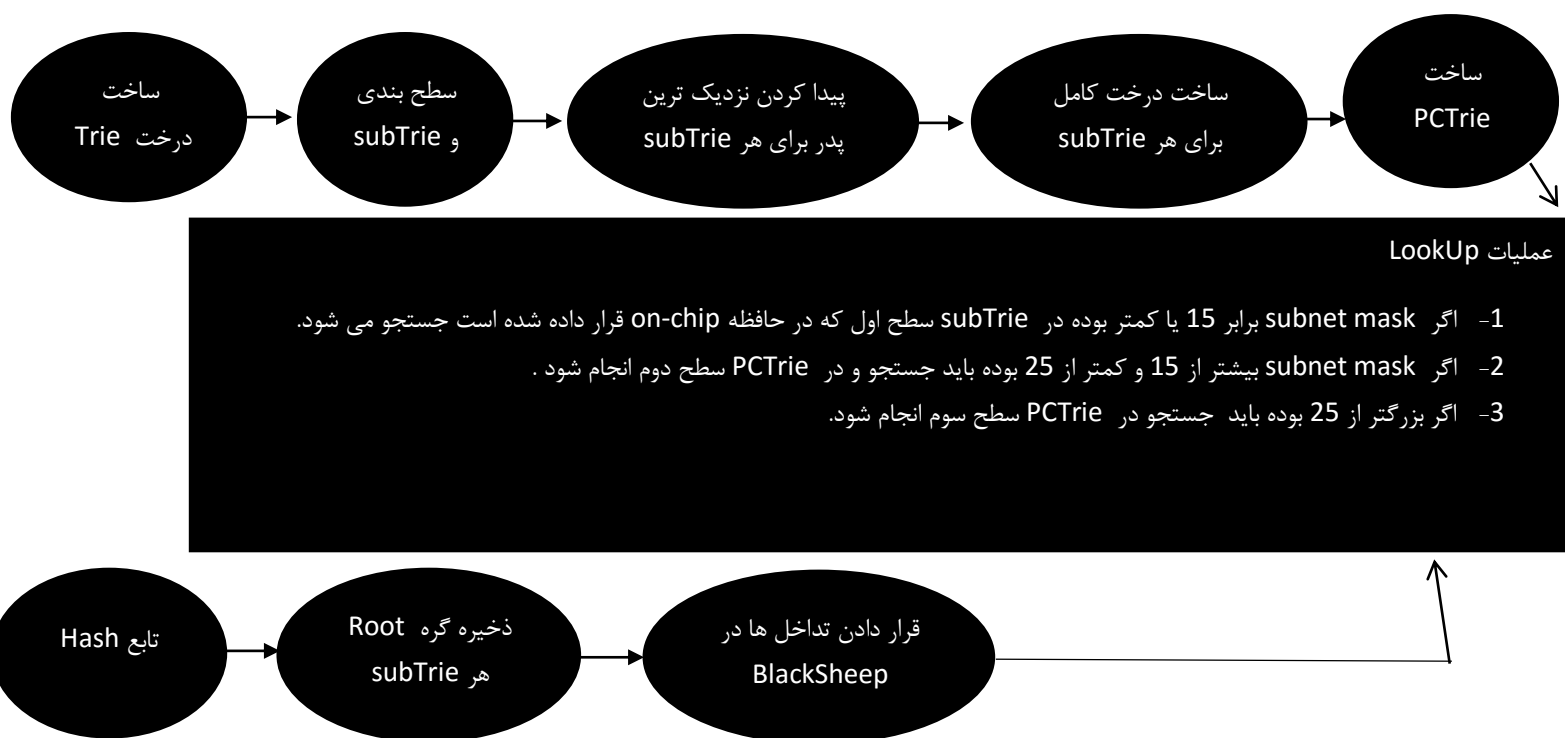
به نام خدا

جابر بابکی 96131020

پروژه اول

بزرگ فکر کن ، هوشمندانه تصمیم بگیر، اما کوچک شروع کن

مسیر پروژه:



شماره	نام تابع	عملکرد
1	createTrie(String key, String next)	این تابع برای ساخت درخت Trie می باشد و با ورودی prefix و next hop درخت trie را می سازد
2	searchInTrie(String key, TrieNode trie)	با گرفتن prefix و درخت trie مورد نظر آن prefix را در آن درخت بررسی می کند
3	printDFS(String prefix, TrieNode n, boolean isLeft)	نمایش درخت Trie به صورت پیمایش عمقی
4	fullTrie(TrieNode trie, int lev)	این تابع یک درخت را می گیرد و آن را کامل می کند.
5	eliminate(TrieNode root)	این تابع یکی از دو شرط ساخت PCTrie را انجام می دهد و node set که دو فرزند داشته باشد را حذف می کند
6	createPCTrie(TrieNode root)	این تابع بعد از انجام دو شرط لازم برای Pctrie این PCTrie را به صورت bitmap و همچنین next hop را در آرایه دو بعدی قرار می دهد
7	hashing(List<String> subTrie, List<TrieNode> subTrieModelLevel, List<TreeBitmapModle> pctrises)	عملیات هشینگ در این تابع انجام می شود و هر گره Root هر subTrie بر اساس تابع hash در جدول hash ذخیره می کند چنانچه برخوردی رخ دهد آن ها را blackSheep ذخیره می کند خروجی این تابع یک memberShip می باشد
8	searchInTreeBitmap(TreeBitmapModle tree, String prefix)	عملیات جستجو در TreeBitmap به این صورت می باشد که ابتدا باید در رشته Bitmap مورد نظر جستجو کرده
9	lookUp(String inputIP)	عملیات lookup در 15 بیت اول
10	convert(String str)	تابع IP را دریافت می کند و به صورت باینری بر اساس subnet mask نشان می دهد.
11	readFromFile()	این تابع فایل ورودی را دریافت و درون اون فایل بر اساس کد های regex که نوشتیم prefix ها و nexthop ها را بیرون می آورد. و خروجی که می دهد یک دیتا مدل PrefixNexthop است.

جدول کلاس های کاربردی

شماره	نام کلاس	عملکرد
1	Trie	در کلاس Trie عملیات توابع ساخت و جستجو و نمایش درخت Trie و همچنین ساخت PCTrie وجود دارد
2	MainLookUp	در این کلاس توابع مهمی همچون hashing و searchInTreeBitmap وجود دارد و به طور کلی عملیات در این کلاس انجام می شود
3	ReadTextFile	کلاسی کاربردی که در آن تابع readFromFile وجود دارد که فایل ها را می خواند و prefix ها را list میکند و نمایش می دهد
4	IpToPrefix	در این تابع یک کلاس وجود دارد آن هم convert هست که Ip ها را به prefix تبدیل می کند

جدول کلاس مدل

شماره	نام مدل	عملکرد
1	TrieNodeModel	این مدل در واقع هر گره در درخت Trie را تشکیل می دهد که شامل فیلد های اشاره گر گره سمت چپ و راست و همچنین prefix و nexthop هست.
2	PCTrie2Modle	این مدل دو فیلد اصلی دارد یکی Treebitmap که در واقع رشته بیت هست و دیگری جدول دو بعدی هست که nexthop را نگه می دارد
3	HashModel	بر طبق member query sheep که در مقاله گفته شده سطر های جدول هش دارای سه فیلد اصلی verifyBit و rootNodeNHI و collision که این ها به عنوان فیلد این دیتا مدل در نظر گرفته شده اند
4	BlackSheepModel	طبق چیزی که در مقاله گفته شده سطر های درون جدول balsckSheep همان فیلد های hashtable هستند با این تفاوت که آدرس PCTrie مستقیم درون آن می باشد
5	MemberShipModel	این دیتا مدل در واقع حاوی HashModel و BlackSheepModel هست و این فیلد را برای تمام سطوح نگه می دارد
6	PrefixNextHob	این دیتا مدل موقع خواندن فایل ها اطلاعات درون این ها قرار می گیرد و در listView که یک کنترل اندرویدی هست نشان داده می شود.

توضیحات و عملکرد هر تابع

1- ساخت درخت Trie

برای ساخت درخت Trie ابتدا باید دیتا مدل ساخت درخت Trie بررسی شود.

```
package com.ario.flashtriebeyond.model;

/**
 * Created by jaberALU on 31/12/2017.
 */

public class TrieNodeModel {
    public final int size = 2;
    public TrieNodeModel[] children = new TrieNodeModel[size];
    public boolean isEndOfPrefix;
    public String next;
    public String pre;

    public TrieNodeModel() {
        isEndOfPrefix = false;
        next = "null";
        pre = "0";
        for (int i = 0; i < size; i++)
            children[i] = null;
    }
}
```

همونطور که در تصویر موجود هست دیتا مدل هر گره از درخت Trie شامل آرایه ای از جنس همان TrieNodeModel هست و در واقع با این کار که انجام دادم با دسترسی به یک گره می توان گره های پایینی آن را هم پیدا کرد سائز این آرایه 2 هست یکی برای فرزند چپ و دیگری برای فرزند راست در واقع گره ها در این درخت را به صورت تو در تو در تشکیل دادم به جز فیلد children سه فیلد دیگه هم وجود دارد فیلد isEndOfPrefix که نشان دهنده این هست که اون گره یک prefix هست یا نه و nexthop و prefix فیلد های دیگر هستند به محض ایجاد شی از این کلاس constructor اون صدا زده می شود و مقدار های پیش فرض را وارد می کند.

حالا تابع ساخت Trie را بررسی می کنیم.

```
public void createTrie(String key, String next) {
    int level;
    int length = key.length();
    int index;
    TrieNodeModel eachNode = root;
    if (key.equals("")) {
        eachNode.isEndOfPrefix = true;
        eachNode.next = next;
        eachNode.pre = key;
    } else {
        char[] arr = key.toCharArray();
        for (level = 0; level < length; level++) {
            index = Integer.parseInt("$" + arr[level]);
            if (eachNode.children[index] == null) {
                eachNode.children[index] = new TrieNodeModel();
            }
            eachNode = eachNode.children[index];
        }
        eachNode.isEndOfPrefix = true;
        eachNode.next = next;
        eachNode.pre = key;
    }
}
```

برای ساخت درخت Trie تابع createTrie صدا زده می شود، که ورودی آن یک key به عنوان prefix و nexthop دریافت می شود، ابتدای تابع یک نود ساخته می شود که با root خود یک نود global در به صورت فیلد در ابتدای کلاس Trie تعریف شده پر می شود، سپس if اول بررسی می کند که اگر ورودی استار هست فیلد های دیگر نود را پر می کند و تابع به پایان می رسد، اگر غیر استار باشد وارد else می شود و بر اساس طول prefix حلقه می چرخد و هر بار بر اساس یک یا صفر بودن فرزند چپ یا راست ایجاد می شود و به همین ترتیب نود جدید ایجاد می شود و در پایان در آخرین نودی که ایجاد شده است next hop و prefix قرار می گیرد و isEndOfPrefix به نشان نودی که حاوی prefix هست پر می شود.

تابع `searchInTrie` در واقع `prefix` و درخت `Trie` داده می شود و سپس آن گره در صورت وجود برمیگرداند اگر آن گره وجود نداشته باشد نزدیکترین گره ای که `prefix` داشته را بر میگرداند .

```
public TrieNodeModel searchInTrie(String key, TrieNodeModel trie) {
    nexLlable = "";
    int level;
    int length = key.length();
    int index;
    TrieNodeModel eachNode = trie;
    if (key.equals("")) {
        return trie;
    } else {
        char[] arr = key.toCharArray();
        for (level = 0; level < length; level++) {
            index = Integer.parseInt( String.valueOf(arr[level]));
            if (!eachNode.next.equals("null")) {
                nexLlable = eachNode.next;
            }
            if (eachNode.children[index] == null)
                return null;
            eachNode = eachNode.children[index];
        }
        return eachNode;
    }
}
```

در ابتدا نودی ساخته می شود و با `Trie` که به عنوان ورودی داده می شود پر می شود با این کار در واقع من هر `subTire` را بخوام میتونم سرچ بزنم توش و همچنین نزدیکترین پدر را پیدا کنم و این تابع خیلی کاربردی هست برای ما و پر کاربرد، یه فیلد استاتیک در کلاس `Trie` قرار دادم که در مراحل پیمایش اگر محتوای نودی غیر `null` بود درونش میذاره و در هر جای دیگه میتونم ازش استفاده کنم کهنشان دهنده نزدیکترین جد یا پدر هست پس با این حرکت در واقع دو تابع را در یک تابع نوشتم.

```
public String printDFS(String prefix, TrieNodeModel n, boolean isLeft) {
    if (n != null) {
        System.out.println(prefix + (isLeft ? "L--> " : "R--> ") + n.next);
        strrr = strrr + "\n" + prefix + (isLeft ? "L--> " : "R--> ") + n.next;
        printDFS(prefix + (isLeft ? "|" : " "), n.children[0], isLeft: true);
        printDFS(prefix + (isLeft ? "|" : " "), n.children[1], isLeft: false);
    }
    return strrr;
}
```

تابع پیمایش به صورت عمقی و بازگشتی نوشته شده است ورودی تابع می تواند یک Trie کلی یا هر subTrie باشد و در هر بار فراخوانی فرزند چپ یا راست که خود یک trie هستند وارد تابع می شود اون prefix که نوشتم یعنی پشت چیزی که نمایش داده میشه یه چیزی بنویسم و نشون داده بشه همچنین من در لاگ سیستم این درخت را نمایش می دهم و در نهایت در یک فیلد استاتیک نمایش را از جنس String هست قرار می دهم و سپس هر جایی که دوست داشتم می خوانم.

4- کامل کردن یک subTrie

```
public void fullTrie(TrieNodeModel trie, int lev) {
    int f = 0;
    if (lev == 0) {
        f = 3;
    } else if (lev == 1) {
        f = 4;
    } else if (lev == 2) {
        f = 15;
    } else if (lev == 3) {
        f = 10;
    } else if (lev == 4) {
        f = 7;
    }
    int i = 0;
    if (trie == null)
        return;
    Queue<TrieNodeModel> q = new LinkedList<TrieNodeModel>();
    q.add(trie);
    while (true) {
        int nodeCount = q.size();
        if (nodeCount == 0 || i >= f)
            break;
        while (nodeCount > 0) {
            TrieNodeModel node = q.peek();
            q.remove();
            if (node.children[0] != null) {
                q.add(node.children[0]);
            } else {
                node.children[0] = new TrieNodeModel();
                q.add(node.children[0]);
            }
        }
        i++;
    }
}
```


در این تابع ورودی یک `subTrie` و `lev` نشان دهنده این مورد هست که تا چه سطحی کامل شود من چون علاوه بر استفاده از فایل که داده شده مثال درون مقاله که تا سطح 3 کامل می کند و همچنین یه مثالی خودم زدم که دو سطحش تا 3 و سطح اخرش تا سطح 4 کامل می شود. از طرفی در مورد IP های واقعی سطح اول از 0 تا 15 سطح دوم از 16 تا 24 و سطح سوم از بیت 25 تا 32 هست پس سطوح متفاوت هست و در ابتدای تابع تعیین شود. سپس گره اول را درون یه صف قرار می دهیم و فرزندان را بررسی می کنیم و چنانچه فرزندان آن `null` باشد برای آن فرزندی ایجاد و آن را درون صف قرار می دهیم این روند تا سطح مورد نظر انجام می شود.

5- ساخت PCTrie

به طور کلی من برای ساخت PCTrie سه مرحله من در نظر گرفتم:

مرحله اول : پیمایش درخت و گره هایی خود `nextHop` هستند ولی برادر آن ها دارای `netxHop` نیست پس باید نزدیکترین جد را پیدا کرده و `nextHop` آن را درونش قرار بدهیم که این خود یک تابع به نام `createPCTrie` شده است

مرحله دوم : این درختی که به صورت بالا درش آوردیم باید `NodeSet` هایی که هر دو فرزند آن ها `nodeSet` بوده را `nextHop` آن ها را پاک بکنیم

مرحله سوم : باید این درخت را `TreeBitmap` بکنیم و `nextHop` آن را درون ارایه دوبعدی قرارا بدهیم

The example shows one subtrie that includes five prefixes ($*, 1*, 00*, 11*, 100*$) and the corresponding NHI (A, B, C, D, E). In *Step 1*, the routing table is simply translated into a binary trie representation. Fig. 3(a) shows Tree Bitmap for the given routing table. Since Tree Bitmap simply converts the binary trie representation to the bitmap representation, *Steps 2* and *3* are the same as in the binary trie. Bit positions in the bitmap are set to “1” at the locations that have a prefix, and set to “0” otherwise, as shown in the *Final Data Structure* in the figure. Now let us look at PC-Trie2 in Fig. 3(b). It illustrates the conversion process from a binary trie to PC-Trie2. The suffix (the number) represents the compression degree. PC-Trie2 means compression of two sibling nodes into one. The two sibling bits that are compressed are marked by a dotted circle. Let us denote this set of nodes as *node sets*.

توجه شود که من از PCTrie 2 استفاده کردم چون مقاله در نهایت بر روی IP ها این PCTrei2 را استفاده کرده است ولی تبدیل به PCTrie 4 یا 8 فقط در مرحله دوم تغییری ایجاد می کند که مقاله این کار را انجام نداده است. به طور کلی PCTrie همان TreeBitmap معمولی هست با این تفاوت که قبلش توسط مرحله اول و دوم فشرده سازی بر روی درخت انجام می دهد و باعث کاهش حافظه مصرفی و افزایش پیچیدگی زمانی می شود. و ایده اصلی مقاله در همین رابطه بوده است.

5-1 مرحله اول: برادر پیدا کردن هر گره بی برادر

در این مرحله با استفاده از تابع مورد نظر تمام گره هایی که خود nestHop دارند ولی برادر آن ها خالی هست را نزدیکترین جد را پیدا کرده و قرار می دهیم.


```

public void createPCTrie(TrieNodeModel root) {
    if (root == null)
        return;
    Queue<TrieNodeModel> q = new LinkedList<TrieNodeModel>();
    q.add(root);
    while (true) {
        int nodeCount = q.size();
        if (nodeCount == 0)
            break;
        while (nodeCount > 0) {
            TrieNodeModel node = q.peek();
            q.remove();
            if (node.children[0] != null) {
                q.add(node.children[0]);
            }
            if (node.children[1] != null) {
                q.add(node.children[1]);
            }

            if (node.children[0] != null && node.children[0].next.equals("null") && node.children[1] != null &&
                !node.children[1].next.equals("null")) {
                if (node.next.equals("null")) {
                    searchInTrie(node.children[1].pre, root);
                    node.children[0].next = nexLlable;
                } else {
                    node.children[0].next = node.next;
                }
            }

            if (node.children[0] != null && !node.children[0].next.equals("null") && node.children[1] != null &&
                node.children[1].next.equals("null")) {
                if (node.next.equals("null")) {
                    searchInTrie(node.children[0].pre, root);
                    node.children[1].next = nexLlable;
                } else {
                    node.children[1].next = node.next;
                }
            }
            nodeCount--;
        }
    }
}

```

همانطور که دیده می شود پیمایش مانند قبلی ها در صف قرار می هد و چک کردن به این صورت است که اگر گره ای غیر null باشد و فرزند چپ مقدار nextHop اش برابر با "null" باشد و فرزند راست غیر null باشد و مقدار nextHop اش برابر غیر "null" باشد یعنی nextHop داشته باشد پس باید نزدیکترین جد برای ان پیدا شود و من در اینجا تابع searchInTrie که گفتم خیلی کاربرد دارد را با prefix برادر صدا می زنم و نزدیکترین پدر پیدا می شود. و قرار می

دهم.

2-5 مرحله دوم: حذف nodeSet ها فرزند دار

```
public void eliminate(TrieNodeModel root) {
    if (root == null)
        return;
    Queue<TrieNodeModel> q = new LinkedList<TrieNodeModel>();
    q.add(root);
    while (true) {
        int nodeCount = q.size();
        if (nodeCount == 0)
            break;
        while (nodeCount > 0) {
            TrieNodeModel node = q.peek();
            q.remove();
            if (node.children[0] != null && node.children[1] != null && !node.children[0].next.equals("null") &&
                !node.children[1].next.equals("null")) {
                if (node.children[0].children[0] != null && node.children[0].children[1] != null &&
                    node.children[1].children[0] != null && node.children[1].children[1] != null &&
                    !node.children[0].children[0].next.equals("null") && !node.children[0].children[1].next.equals("null") &&
                    !node.children[1].children[0].next.equals("null") && !node.children[1].children[1].next.equals("null")) {
                    node.children[0].next = "null";
                    node.children[1].next = "null";
                }
            }
            if (node.children[0] != null) {
                q.add(node.children[0]);
            }
            if (node.children[1] != null) {
                q.add(node.children[1]);
            }
            nodeCount--;
        }
    }
}
```

در این تابع با بررسی اینکه یک nodeSet فرزند چپ و راست داشته باشد در این صورت nextHop خودشون را برابر null قرار می دهیم.

3-5 مرحله سوم: ساخت TreeBitmap

در این مرحله باید رشته بیتی به همراه جدول nextHop ها ایجاد شود، تفاوتی که در این مرحله وجود دارد این هست که ما دیتا مدل را عوض کردیم و دیگر باید با دیتا مدل جدید کار کنیم که PCTrie2Modle که ابتدا این را بررسی میکنیم:

```
package com.ario.flashtriebeyond.model;

/**
 * Created by jaberALU on 31/12/2017.
 */

public class PCTrie2Modle {
    public String[][] children = new String[1100][2];
    public String bitmap="";
    public String root="";
}
```

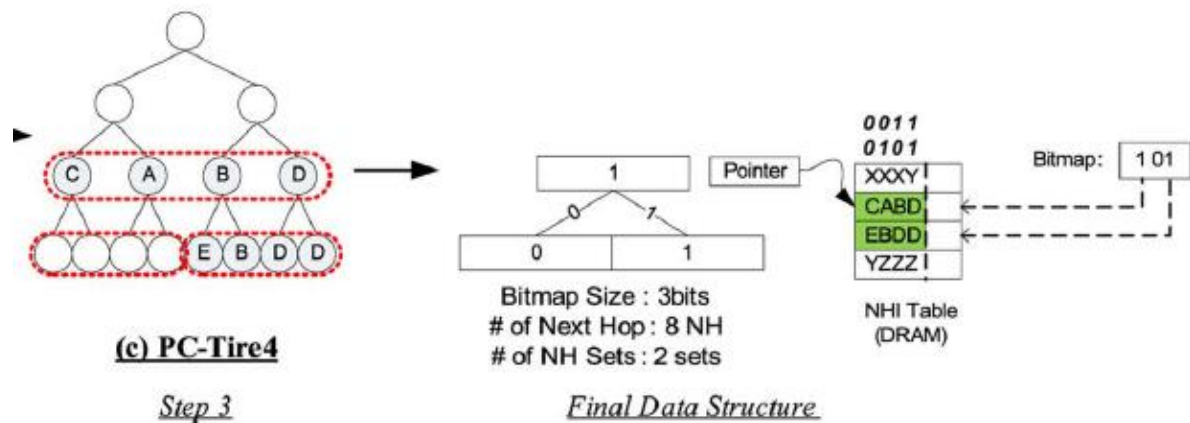
در این دیتا مدل ما یک ارایه دوبعدی داریم که برحسب که 1100 سطر به دلیل نصف بودن تعداد TreeBitmap ها سطح واقعی IP و دو ستون که نشان دهنده هر nodeSet می باشد و همچنین یک bitmap که رشته صفرو یک ها می باشد .

نکته ای که وجود دارد این هست که اگر subTrie فقط از یک ریشه تشکیل شده باشد و هیچ گره دیگری در آن subTrie وجود نداشته باشد در طی مراحل سوم اون گره حذف می شود به همین دلیل ما این جا یک فیلد root قرار دادیم که ریشه هر subTrie را قرار می دهد تا پس از PCTrie شدن گم نشود.

حالا به بررسی عملیات PCTrie در مرحله سوم می پردازیم:

```
public PCTrie2Module treeBitmap(TrieNodeModel root) {
    PCTrie2Module PC = new PCTrie2Module();
    int i = 0;
    int j = 0;
    PC.root = root.next;
    root.next = "null";
    if (root == null)
        return PC;
    Queue<TrieNodeModel> q = new LinkedList<TrieNodeModel>();
    q.add(root);
    while (true) {
        int nodeCount = q.size();
        if (nodeCount == 0)
            break;
        while (nodeCount > 0) {
            TrieNodeModel node = q.peek();
            q.remove();
            if (node.children[0] != null && node.children[1] != null) {
                if (!node.children[0].next.equals("null") && !node.children[1].next.equals("null")) {
                    j = 0;
                    PC.bitmap = PC.bitmap & "1";
                    PC.children[i][j] = node.children[0].next;
                    j++;
                    PC.children[i][j] = node.children[1].next;
                    i++;
                } else if (node.children[0].next.equals("null") && node.children[1].next.equals("null")) {
                    PC.bitmap = PC.bitmap & "0";
                }
            }
            if (node.children[0] != null)
                q.add(node.children[0]);
            if (node.children[1] != null)
                q.add(node.children[1]);
            nodeCount--;
        }
        System.out.println();
    }
    return PC;
}
```

در این تابع اگر گره ای فرزند چپ و راست داشته باشد که هر دو nextHop داشته باشند به ترتیب آن ها را در فیلد ارایه دوبعدی children ذخیره می کنیم با استفاده از شمارنده های zr و همچنین در فیلد bitmap 1 می گذاریم، اگر مخالف این حالت وجود داشته باشد فقط در bitmap صفر قرار می دهیم



6- تابع هشینگ

تا به اینجای کار ما PCTrie را انجام دادیم، و حالا باید عملیات هشینگ و سپس lookup را انجام دهیم بر طبق چیزی که مقاله گفته شده ما باید آدرس ریشه هر subTrie کلش یا بخش از آن را به عنوان کلید به تابع هش بدهیم و ادرس ریشه را بر اساس خروجی تابع هش درون جدول هش بر اساس ساختار سطر که گفته است ذخیره کنیم چنانچه تابع هش خروجی داده است که قبلا با آن کلید سطری در جدول هش وجود داشته باشد باید آن سطر جدید را در blackSheep مموری با ساختار سطر جدید ذخیره بکنیم.

with a hash function. Each entry of the hash table holds all or a portion of the root IP address of the programmed subtrie. We call this entry *verify bits* and perform an exact matching operation with the input IP address. Hash functions inherently do not have any false negatives. By means of an exact matching operation, we ensure no false positives as well.

possible collisions. Therefore, the hash table has two types of entries: one each for collision and noncollision cases as shown in the figure. If the hash table entry has a collision, then its Least Significant Bit (LSB) is set to “1”; else, it is set to “0” for no collision. The collided items are stored in Black Sheep (BS) memory located in the membership query module.

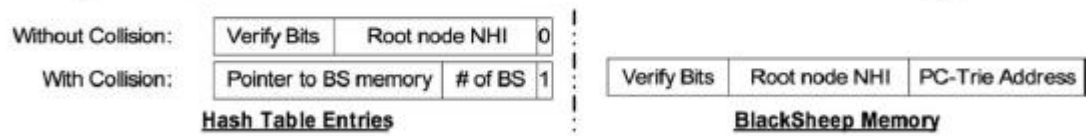


Fig. 4. Block diagram of membership query module.

بر طبق تصویر موجود ما دو نوع سطر در جدول داریم که یکی verify Bits و Rootnode NHI و 0 و دیگری pointer BS و 1 موقعه ای که برخوردی رخ دهد ما باید درون سطر جدول هش سطر با بیت 1 قرار دهیم و آن را به BlackSheep ببریم و که انجا هم ساختار خودش را دارد و verifyBit و Root node NHI و آدرس PCTrie به طور مستقیم قرار دارد. ما برای هر سطح این جدول ها را داریم

برای اینکه عملیات هشینگ به طور واضح تر انجام شود ابتدا دیتا مدل مورد استفاد را بررسی کنیم

```
package com.ario.flashtriebeyond.model;

/**
 * Created by jaberALU on 20/01/2018.
 */

public class HashModel {
    private String verifyBit;
    private String rootNodeNHI;
    private Integer collision;
    public HashModel(String verifyBit, String rootNodeNHI, Integer collision) {
        this.verifyBit = verifyBit;
        this.rootNodeNHI = rootNodeNHI;
        this.collision = collision;
    }
    public String getVerifyBit() { return verifyBit; }

    public String getRootNodeNHI() { return rootNodeNHI; }

    public Integer getCollision() { return collision; }
    public void setVerifyBit(String verifyBit) { this.verifyBit = verifyBit; }

    public void setRootNodeNHI(String rootNodeNHI) { this.rootNodeNHI = rootNodeNHI; }

    public void setCollision(Integer collision) { this.collision = collision; }
}
```


دیتا مدلی که برای هر سطر جدول هش استفاده می شود، و دقیقاً همان فیلدهایی هست که در مقاله گفته شده است به همراه setter و getter مربوط به آن‌ها.

```
package com.ario.flashtriebeyond.model;

/**
 * Created by jaberALU on 20/01/2018.
 */

public class BlackSheepModel {
    private String verifyBit;
    private String rootNodeNHI;
    private PCTrie2Modle trieNode;
    public BlackSheepModel(String verifyBit, String rootNodeNHI, PCTrie2Modle trieNode) {
        this.verifyBit = verifyBit;
        this.rootNodeNHI = rootNodeNHI;
        this.trieNode = trieNode;
    }
    public String getVerifyBit() { return verifyBit; }
    public String getRootNodeNHI() { return rootNodeNHI; }
    public PCTrie2Modle getTrieNode() { return trieNode; }
    public void setVerifyBit(String verifyBit) { this.verifyBit = verifyBit; }
    public void setRootNodeNHI(String rootNodeNHI) { this.rootNodeNHI = rootNodeNHI; }
    public void setTrieNode(PCTrie2Modle trieNode) { this.trieNode = trieNode; }
}
```

این هم دیتا مدل مربوط به blackSheep تنها فرقی که دارد این که فیلدی مستقیم به PCTrie اشاره می کند .

```
package com.ario.flashtriebeyond.model;

import java.util.Hashtable;
import java.util.List;

/**
 * Created by jaberALU on 20/01/2018.
 */

public class MemberShipModel {
    public Hashtable<Integer, HashModel> hashTable;
    public List<BlackSheepModel> BlackSheep;
    public Hashtable<Integer, HashModel> getHashTable() { return hashTable; }
    public List<BlackSheepModel> getBlackSheep() { return BlackSheep; }
    public void setHashTable(Hashtable<Integer, HashModel> hashTable) { this.hashTable = hashTable; }
    public void setBlackSheep(List<BlackSheepModel> blackSheep) { BlackSheep = blackSheep; }
}
```

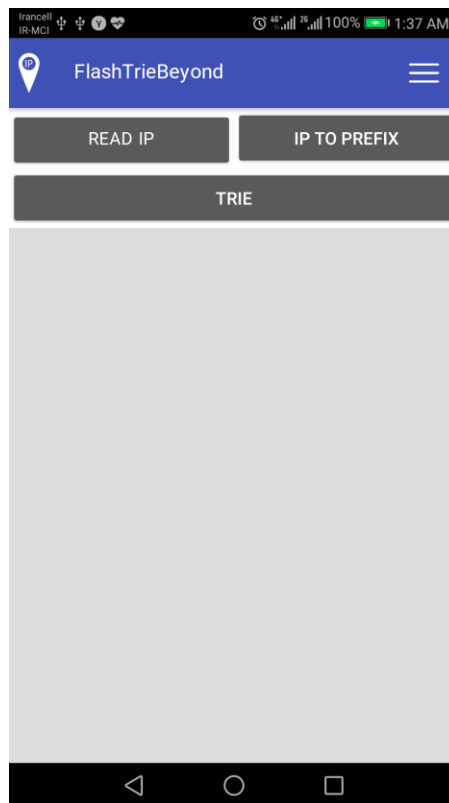
این دیتا مدل memberShip در واقع همه hashTable و BlackSheep ها را درون خود نگه می دارد،

حالا که با دیتا مدل آشنا شدیم می پردازیم به عملکرد تابع .

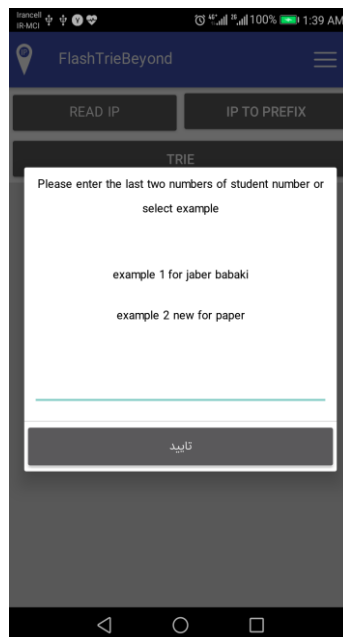
```
public MemberShipModel hashing(List<String> subTrie, List<TrieNodeModel> subTrieModelLevel, List<PCTrie2Modle> pctries) {
    Hashtable<Integer, HashModel> hashTable2 = new Hashtable<Integer, HashModel>();
    List<BlackSheepModel> BlackSheep2 = new ArrayList<BlackSheepModel>();
    for (int i = 0; i < subTrie.size(); i++) {
        int thk = (Integer.parseInt(subTrie.get(i), radix: 2)) % 4;
        HashModel hashModel = hashTable2.get(thk);
        if (hashModel == null) {
            hashTable2.put(thk, new HashModel(subTrie.get(i), subTrieModelLevel.get(i).next, collision: 0));
        } else {
            BlackSheepModel blackSheepModel = new BlackSheepModel(subTrie.get(i), subTrieModelLevel.get(i).next, pctries.get(i));
            if (hashModel.getCollision() == 0) {
                BlackSheep2.add(blackSheepModel);
                BlackSheepModel blackSheepModel2 = new BlackSheepModel(hashModel.getVerifyBit(), hashModel.getRootNodeNHI(), trieNode: null);
                BlackSheep2.add(blackSheepModel2);
                hashTable2.put(thk, new HashModel(verifyBit: "!", rootNodeNHI: "" + ((BlackSheep2.size() - 1) + ", " +
                    (BlackSheep2.size() - 2)), collision: 1));
            } else {
                BlackSheep2.add(blackSheepModel);
                hashTable2.put(thk, new HashModel(verifyBit: "!", rootNodeNHI: hashModel.getRootNodeNHI() + ", " +
                    (BlackSheep2.size() - 1), collision: 1));
            }
        }
    }
    MemberShipModel memberShip = new MemberShipModel();
    memberShip.setBlackSheep(BlackSheep2);
    memberShip.setHashTable(hashTable2);
    return memberShip;
}
```

در اینجا ما می اییم طبق چیزی که گفتیم عمل می کنیم یعنی ریشه هر suntrie را با تابع هش ساده ی $K \bmod M$ کلیدی را بدست می آوریم، و جای اون کلید در hashtable بررسی می کنیم اگر قبلا داده ای بوده پس این داده جدید به همراه داده قبلی باید وارد blackSheep شوند و در hashtable اشاره گری به اون وجود داشته باشد که من در اینجا اندیس list را ذخیره می کنم به عنوان اشاره گر به blackSheep و در نهایت این HshTable و List از جنس BlackSheep به عنوان MemebrShip برمی گردانم.

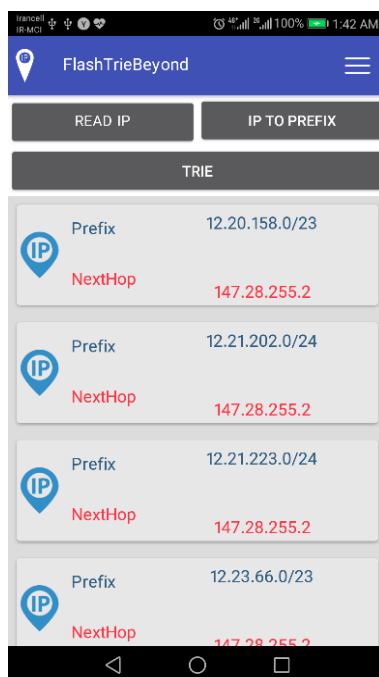
انجام عملیات lookup



در ابتدا که برنامه اجرا می شود عملیاتی انجام نمیشود و صفحه به صورت بالا هست، سپس با انتخاب گزینه Read Ip صفحه برای انتخاب IP باز می شود.



در اینجا می توان مثال مقاله را انتخاب کرد یا مثال که خودم زدم و یا اینکه شماره دانشجویی بدهیم و بر اساس فرمول گفته شده 100 تا خط انتخا و prefix و nextHop ها لیست شود.

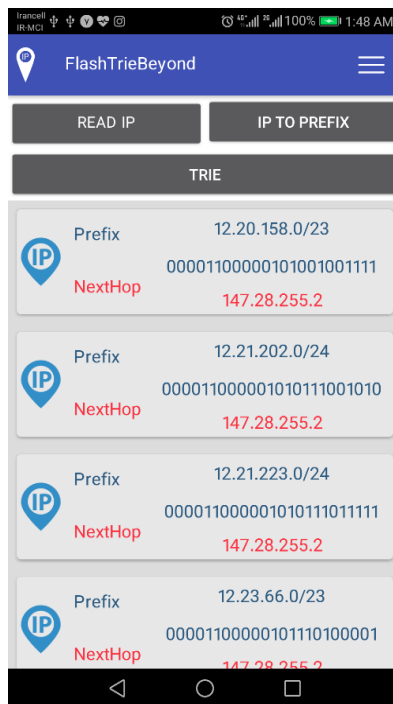


با وارد کردن عدد 20 به عنوان دو شماره آخر دانشجویی لیست زیر انتخاب می شود یعنی تابع `readFromFile` فراخوانی می شود که عملکردش به شرح زیر است

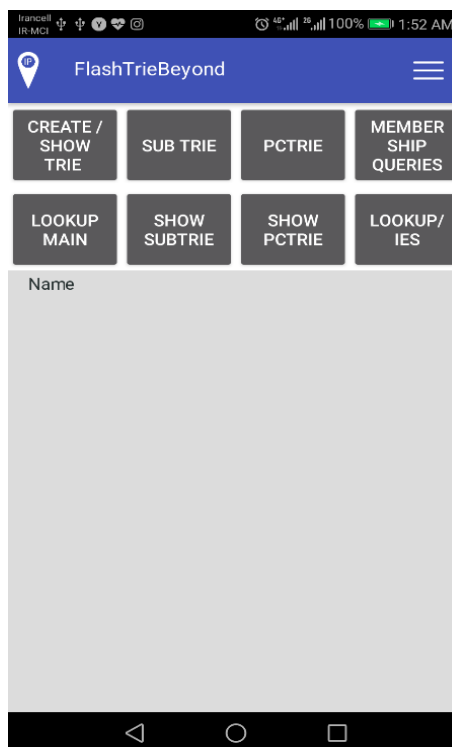
```
public List<PrefixNextHop> readFromFile() {
    String returnString = "\n";
    InputStream fIn = null;
    InputStreamReader isr = null;
    BufferedReader input = null;
    try {
        fIn = getContext().getResources().getAssets().open("data/" + getFileName(), Context.MODE_WORLD_READABLE);
        isr = new InputStreamReader(fIn);
        input = new BufferedReader(isr);
        String line = "";
        int counter = 0;
        int m = 0;
        Log.i("tag: TXT", "msg: " + getSelectLine());
        while ((line = input.readLine()) != null) {
            counter++;
            if (counter == selectLine && m <= 100) {
                selectLine = selectLine + 10;
                Pattern p = Pattern.compile("(?<=\\broute\\b).*?(?=\\borigin\\b)");
                Matcher m1 = p.matcher(line);
                PrefixNextHop pre = new PrefixNextHop();
                while (m1.find()) {
                    pre.setEnterPort(m1.group().trim());
                }
                pre.setNextHop(line.substring((line.indexOf("next-hop") + 8), line.length()));
                m++;
                returnString = returnString + (line + "\n");
                if (m >= 100) {
                    break;
                }
            }
            prefixNextHop.add(pre);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

همانطور که نشان داده می شود تابع خواندن فایل که خط به خط خوانده و بر اساس الگویی که دادم 100 خط خوانده و درون لیست قرار می دهد و سپس نمایش می دهد.

سپس باید گزینه IpToPrefix را بزنیم تا IP ها به Prefix ها تبدیل شوند و در شکل زیر نشان داده می شود



حال می توان با زدن گزینه Trie وارد صفحه Trie شد که به شکل زیر می باشد .



کدهایی که در این صفحه وجود دارد به این صورت هست

```
btnLookupTrie.setOnClickListener(new View.OnClickListener() {(...)});
btnShowTrie.setOnClickListener(new View.OnClickListener() {(...)});
btnSubTrieCreate.setOnClickListener(new View.OnClickListener() {(...)});
btnSubTrieShow.setOnClickListener(new View.OnClickListener() {(...)});
btnPCTrieCreate.setOnClickListener(new View.OnClickListener() {(...)});
btnPCTrieShow.setOnClickListener(new View.OnClickListener() {(...)});
btnHash.setOnClickListener(new View.OnClickListener() {(...)});
btnMember.setOnClickListener(new View.OnClickListener() {(...)});
}

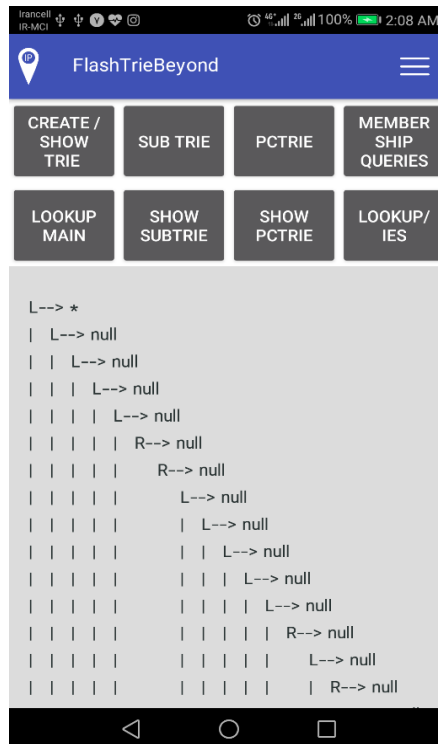
public boolean contain(List<String> arr, String item) {...}
public void clearShow() {(...)}
public void clearAllArrayList() {...}
public void clearArrayList() {...}
public void clearState() {...}
public PCTrie2Modle getPCTrieIndex(List<String> subTrie, String item, List<PCTrie2Modle> pctries) {...}
public String printAllPCTrie(List<PCTrie2Modle> pctries) {(...)}
public String printPCTrie(PCTrie2Modle pctries) {(...)}
public String lookUp(String inputIP) {...}
public MembershipModel hashing(List<String> subTrie, List<TrieNodeModel> subTrieModelLevel, List<PCTrie2Modle> pctries) {...}
public String searchInTreeBitmap(PCTrie2Modle tree, String prefix) {...}
```

که مشاهده می شود هر دکمه برای خورش اکشنی دارد و به همراه یکسری توابع کاردی .

موقعه ای که کاربر بروی دکمه CreateTrie انتخاب می کند اکشن آن که به صورت زیر است اجر می شود:

```
btnShowTrie.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        clearAllArrayList();
        clearShow();
        clearState();
        stateLookup = 1;
        trieRoot = new Trie();
        for (int i = 0; i < keys.length; i++) {
            trieRoot.createTrie(keys[i], next[i]);
        }
        txtShowTrie.setText(trieRoot.printDFS( prefix: "", trieRoot.root, isleft: true));
    }
});
```

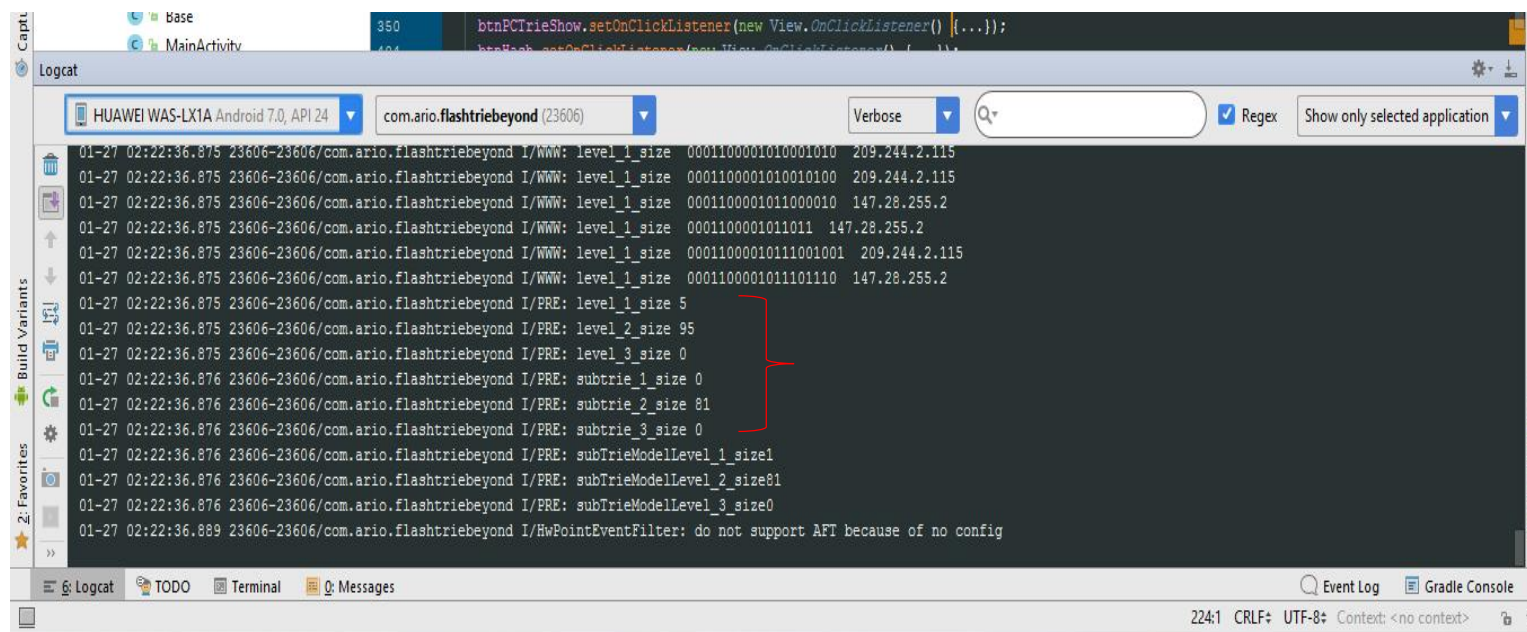
همانطور که دیده می شود بر اساس طول آرایه prefix ها حلقه میچرخد و تابع createTrie که بالاتر فراخوانی کردیم ساخته میشود و همه این ها درون شی trieRoot هست و در نهایت با استفاده از printDFS نمایش داده می شود که به صورت زی است .



که میتوان اسکرول کرد و کل درخت را ببینیم ، در مرحله بعد subTrie باید subtrie بسازیم ایده ای که برای ساخت subTrie استفاده کردم به این صورت می باشد که درخت را هر بار تا یک سطحی بسازیم و از انجایی می دانیم هر سطح چند بیت است میتوانیم جستجویی تا اول هر سطح بنزیم و گره موردنظر که حاوی زیردرخت های خود است را بگیریم و درون لیست سطح دوم قرار دهیم . به صورت زیر :

```
btnSubTrieCreate.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        if (stateLookup == 1) {
            clearArrayList();
            statePCTrie = 0;
            trielevel0 = new Trie();
            for (int i = 0; i < keys.length; i++) {
                if (keys[i].length() <= Base.level0) {
                    level1.add(keys[i]);
                    trielevel0.createTrie(keys[i], next[i]);
                }
            }
            subTrieModelLevel1.add(trielevel0.searchInTrie(keys[0], trieRoot.root));
            trieleveland2 = new Trie();
            for (int i = 0; i < keys.length; i++) {
                if (keys[i].length() <= Base.level0) {
                    trieleveland2.createTrie(keys[i], next[i]);
                } else if (keys[i].length() >= Base.level1L && keys[i].length() <= Base.level1T) {
                    trieleveland2.createTrie(keys[i], next[i]);
                }
            }
            for (int i = 0; i < keys.length; i++) {
                if (keys[i].length() >= Base.level1L && keys[i].length() <= Base.level1T) {
                    level2.add(keys[i]);
                    if (contain(subTrie2, level2.get(level2.size() - 1).substring(0, Base.level1L))) {
                        subTrie2.add(level2.get(level2.size() - 1).substring(0, Base.level1L));
                        subTrieModelLevel2.add(trieleveland2.searchInTrie(subTrie2.get(subTrie2.size() - 1), trieleveland2.root));
                    }
                }
            }
        }
    }
});
```

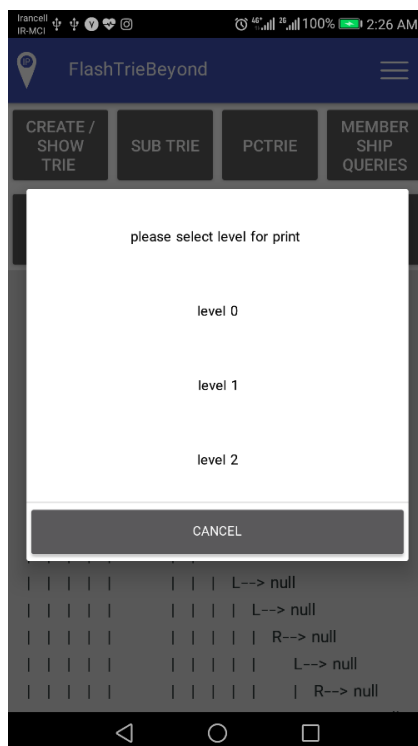
همونطور که مشخص است درخت را تا سطح مورد نظر می سازیم و جستجویی تا ابتدای اون سطح که میخواییم برش دهیم بر اساس بیت ها جدا میکنیم و درون لیست `subTrieModelLevel` قرار می دهیم بر اساسی چیزه که در مقاله گفته باید در لیستی جدا نگه داری شوند.



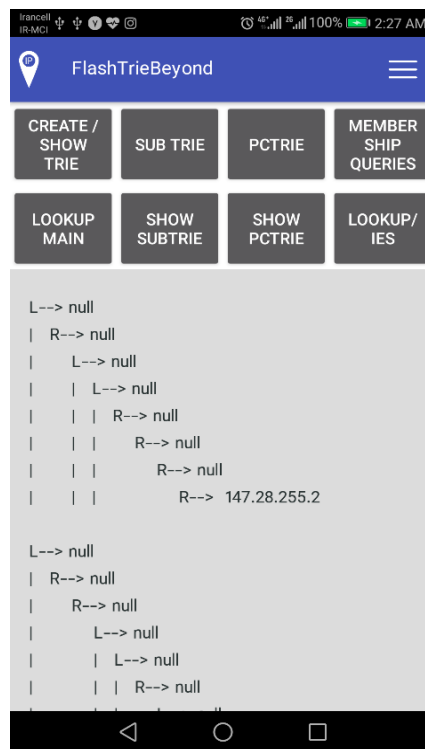
```
01-27 02:22:36.875 23606-23606/com.ario.flashtriebeyond I/WWW: level_1_size 0001100001010001010 209.244.2.115
01-27 02:22:36.875 23606-23606/com.ario.flashtriebeyond I/WWW: level_1_size 0001100001010010100 209.244.2.115
01-27 02:22:36.875 23606-23606/com.ario.flashtriebeyond I/WWW: level_1_size 0001100001011000010 147.28.255.2
01-27 02:22:36.875 23606-23606/com.ario.flashtriebeyond I/WWW: level_1_size 0001100001011011 147.28.255.2
01-27 02:22:36.875 23606-23606/com.ario.flashtriebeyond I/WWW: level_1_size 00011000010111001001 209.244.2.115
01-27 02:22:36.875 23606-23606/com.ario.flashtriebeyond I/WWW: level_1_size 0001100001011101110 147.28.255.2
01-27 02:22:36.875 23606-23606/com.ario.flashtriebeyond I/PRE: level_1_size 5
01-27 02:22:36.875 23606-23606/com.ario.flashtriebeyond I/PRE: level_2_size 95
01-27 02:22:36.875 23606-23606/com.ario.flashtriebeyond I/PRE: level_3_size 0
01-27 02:22:36.876 23606-23606/com.ario.flashtriebeyond I/PRE: subtrie_1_size 0
01-27 02:22:36.876 23606-23606/com.ario.flashtriebeyond I/PRE: subtrie_2_size 81
01-27 02:22:36.876 23606-23606/com.ario.flashtriebeyond I/PRE: subtrie_3_size 0
01-27 02:22:36.876 23606-23606/com.ario.flashtriebeyond I/PRE: subTrieModelLevel_1_size1
01-27 02:22:36.876 23606-23606/com.ario.flashtriebeyond I/PRE: subTrieModelLevel_2_size81
01-27 02:22:36.876 23606-23606/com.ario.flashtriebeyond I/PRE: subTrieModelLevel_3_size0
01-27 02:22:36.889 23606-23606/com.ario.flashtriebeyond I/HwPointEventFilter: do not support AFT because of no config
```

این خروجی لاگ محیط اندروید استدیو هست که نشان می دهد در هر سطح ما چند تا `subTrie` داریم.

حال اگر `subTrieShow` را انتخاب کنیم ابتدا باید سطحی که میخواییم `subTrie` هاش نمایش داده شود را انتخاب کنیم.



با فرض انتخاب سطح اول هر subTrie را به تابع PrintDfs میدهم و خروجی به صورت زیر است.



حال باید عملیات Pctrie بروی هر subTrie انجام شود با انتخاب گزینه create PCTrie اکشن زیر انجام می شود.

```
//if not root find parent
for (int i = 0; i < subTrieModelLevel2.size(); i++) {
    if (subTrieModelLevel2.get(i).next.equals("null")) {
        trielevel1and2.searchInTrie(subTrie2.get(i), trielevel1and2.root);
        subTrieModelLevel2.get(i).next = trielevel1and2.nextLlable;
    }
}

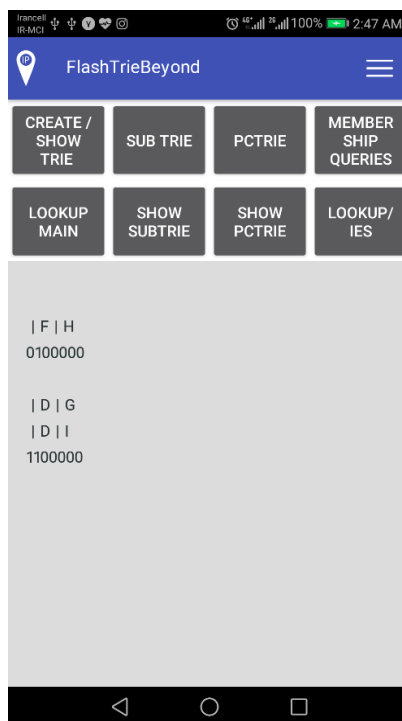
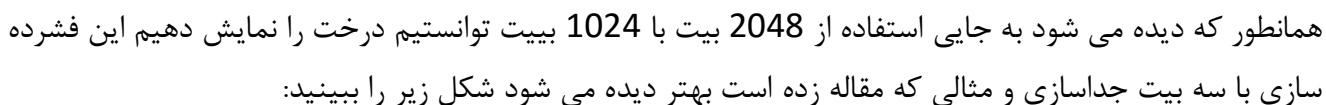
for (int i = 0; i < subTrieModelLevel3.size(); i++) {
    if (subTrieModelLevel3.get(i).next.equals("null")) {
        trielevel1and2.searchInTrie(subTrie3.get(i), trielevel1and2.root);
        subTrieModelLevel3.get(i).next = trielevel1and2.nextLlable;
    }
}

//fullTrie
trielevel0.fullTrie(subTrieModelLevel1.get(0), Base.limit0);
for (int i = 0; i < subTrieModelLevel2.size(); i++) {
    trielevel1and2.fullTrie(subTrieModelLevel2.get(i), Base.limit1);
}

for (int i = 0; i < subTrieModelLevel3.size(); i++) {
    trielevel1and2.fullTrie(subTrieModelLevel3.get(i), Base.limit2);
}

//PCTrie
for (int i = 0; i < subTrieModelLevel2.size(); i++) {
    trielevel1and2.createPCTrie(subTrieModelLevel2.get(i));
    trielevel0.eliminate(subTrieModelLevel2.get(i));
    pctrises2.add(trielevel1and2.treeBitmap(subTrieModelLevel2.get(i)));
}
}
```


سپس عملیات کامل کردن هر subTrie انجام شود و در نهایت اون سه مرحله Pctrie برای هر subTrie فراخوانی می شود و درون لیست pctrie ها قرار می گیرد. حالا اگر نمایش PCTrie را بنویسیم و یکی از سطوح را انتخاب کنیم شکل زیر را مشاهده می کنیم :



شکل بالا PCTrie مثال درون مقاله هست که با در حالی که با 15 بیت TreeBitmap معمولی را نمایش میدادند در این مقاله و با استفاده از این ایده توانستیم با هفت بیت درخت را نمایش دهیم .

حالا باید عملیات memberQuery و lookup را انجام دهیم،
موقع ای که کاربر دکمه memberSheep را می زند اکشن زیر اتفاق می افتد.

```
btnHash.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        if (statePCTrie == 1) {

            MemberShip2 = new MemberShipModel();
            MemberShip2 = hashing(subTrie2, subTrieModelLevel2, pctrises2);

            MemberShip3 = new MemberShipModel();
            MemberShip3 = hashing(subTrie3, subTrieModelLevel3, pctrises3);
            |
            Toast.makeText(context: MainLookUp.this, text: "applying Hashing ", Toast.LENGTH_LONG).show();
            stateHash = 1;
        } else {
            Toast.makeText(context: MainLookUp.this, text: "please do PCTrie ", Toast.LENGTH_LONG).show();
        }
    }
});
```

برای هر سطح تابع hashing صدا زده می شود و خروجی memberQuery برمی گرداند که حاوی جدول هش و BlackSheep هست عملیات hashing بالاتر مفصل توضیح داده شده است.
و حالا عملیات lookup موقعه ای که عملیات lookup زده می شود بر اساس تصویر زیر سه حالت ممکن است رخ دهد

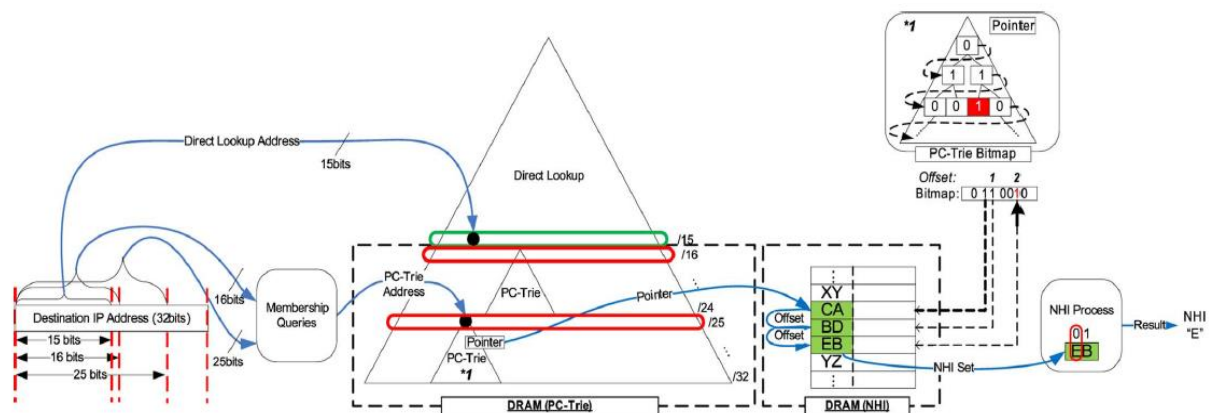


Fig. 8. FlashTrie architecture for IPv4.

حالت اول اگر subnet 15 بیت یا کمتر باشد در on-Chip جستجو انجام می شود و خروجی داده می شود که کدش را من به صورت زیر نوشتم

```

public String lookUp(String inputIP) {
    String str = "";
    TrieNodeModel pre = trieRoot.searchInTrie(inputIP, trieRoot.root);
    if (pre != null && !pre.next.equals("null")) {
        return pre.next;
    } else if (pre != null && pre.next.equals("null")) {
        if (!Trie.nexLlable.equals("")) {
            return Trie.nexLlable;
        }
    } else if (pre == null) {
        if (!Trie.nexLlable.equals("")) {
            return Trie.nexLlable;
        }
    }
    return str;
}

```

که یک سرچ معمولی هست و من IP ورودی را به Prefix تبدیل و بر اساس subnet جدا کرده و به تابع بالا می دهیم و تابع searchInTrie برای سطح اول فراخوانی و حالت های مختلف نتیجه بررسی و خروجی داده می شود اما برای حالت های 15 تا 24 بیت و 25 تا 32 بیت بیاد روند زیر انجام شود:

ابتدا باید بر اساس subnet برش داده شود و سپس به تابع هش داده شود خروجی تابع هش را در hashtable جستجو کنیم وسطر مورد نظر را پیدا کنیم اگر بیت اخر سطر یا همون collision برابر صفر بود حالا باید بگردیم بر اساس verify مورد نظر Pctrie متناظر را پیدا کنیم و سپس درون Pctrie مورد نظر ابتدا Bitmap را بررسی کنیم تا ببینیم در محل مورد نظر 1 وجود دارد یا صفر اگر یک بود حالا باید آرایه دوبعدی من نظر ب با اندیس تعداد یک هایی که در Bitmap رد کردیم جستجو کنیم و بر اساس صفر یا یک بودن بیت کم ارزش محتوا خانه 0 یا 1 آرایه را به عنوان خروجی بدهیم اگر collision برابر یک بود باید برویم در BlackSheep بگردیم و در اونجا خوشبختانه آدرس Pctrie را نگه داشتیم و مستقیم باید همان عملیات جستجو درون Pctrie را انجام بدهیم .

The input 32-bit IPv4 address is categorized in IPv4/15, IPv4/16, and IPv4/25. IPv4/15 is resolved using Direct Lookup (on-chip), and IPv4/16 and IPv4/25 are resolved using the membership query module (as explained in Section III-C.1) (on-chip) and PC-Trie (off-chip). We resolve the PC-Trie address, marked *1 in the figure, from the output of the membership query module. Suppose "100" is the input to this PC-Trie. We can simply traverse the PC-Trie bitmap as we do in a binary trie. If the bit is "0," then we go to the left child; else we go to the right child. We start traversing from the MSB of input. The aim is to find the longest matching prefix in the PC-Trie. After traversing "1" (Right) and "0" (Left), we end up with the third bit in the bottom (dark square in the PC-Trie). The content of the bitmap is "1," which means NHI exists for the input "100." Since this is the longest matching prefix, we resolve the address of NHI memory for this node set. The

```

String prefix = inputIP.substring(0, 8);
int thk = (Integer.parseInt(prefix, radix: 2)) % 4;
Hashtable<Integer, HashModel> hash3 = MemberShip3.getHashTable();
List<BlackSheepModel> blackSheep = MemberShip3.getBlackSheep();
HashModel hModel = hash3.get(thk);
if (hModel.getCollision() == 0) {
    String nextHopRoot = hModel.getVerifyBit();
    txtShowTrie.setText("" + printPCTrie(getPCTrieIndex(subTrie3, nextHopRoot, potries3));
} else if (hModel.getCollision() == 1) {
    String nextHopRoot = hModel.getRootNodeNHI();
    String[] str = nextHopRoot.split( regex: ",");
    for (int i = 0; i < str.length; i++) {
        if (blackSheep.get(Integer.parseInt(str[i])).getVerifyBit().equals(prefix)) {
            if (blackSheep.get(Integer.parseInt(str[i])).getTrieNode() == null) {
                PCTrie2Modle tree = getPCTrieIndex(subTrie3, blackSheep.get(Integer.parseInt(str[i])).getVerifyBit(), potries3);
                String a = lookUp(inputIP);
                a=searchInTreeBitmap(tree, inputIP);
                txtShowTrie.setText("" + printPCTrie(tree) + "\n" + a);
            } else {
                txtShowTrie.setText("" + printPCTrie(blackSheep.get(Integer.parseInt(str[i])).getTrieNode());
            }
        }
        break;
    }
}
}
}

```

این عملیات برای بررسی کردن برخورد و در نهایت بدست آوردن آدرس PCTrie می باشد.

در ابتدا prefix به تابع هش داده می شود و بر اساس خروجی getCollision بررسی می شود اگر صفر بود فقط کافیه با getVerifyBit بتوانیم index اون PCTrie را بدست بیاوریم که این کار توسط تابع getPCTrieIndex انجام می شود اگر برخورد داشت حالا باید بریم BlackSheep جستجو کنیم ولی از اونجایی که اندیس جاهایی که در BlackSheep باید بگردیم را درون getRootNodeNHI قرار دادیم فقط همان خونه ها در BlackSheep را جستجو می کنیم و بایه گام کمتر به PCTrie می رسیم حالا باید در PCTrie عملیات جستجو را انجام دهیم .

```

public String searchInTreeBitmap(PCTrie2Modle tree, String prefix) {
    String result = "";
    String bitmap = prefix.substring(8, prefix.length() - 1);
    String NHI = prefix.substring(prefix.length() - 1, prefix.length());
    int m = 0;
    for (int i = 0; i < tree.bitmap.length(); i++) {
        String h = tree.bitmap.substring(0, 1);
        for (int f = 0; f <= 1; f++) {
            if (tree.children[i][f] != null) {
                if (h.equals("1")) {
                    m = m++;
                }
            }
        }
    }
    if (NHI.equals("0")) {
        result = tree.children[m][0];
    } else {
        result = tree.children[m][1];
    }
    return result;
}

```


درون PCTrie مورد نظر ابتدا Bitmap را بررسی کنیم تا ببینیم در محل مورد نظر 1 وجود دارد یا صفر اگر یک بود حالا باید آرایه دوبعدی منظر ب با اندیس تعداد یک هایی که در Bitmap رد کردیم جستجو کنیم و بر اساس صفر یا یک بودن بیت کم ارزش محتوا خانه 0 یا 1 آرایه را به عنوان خروجی و در نهایت nextHop مورد نظر یافت می شود

خروجی این مرحله به صورت زیر است :

