2021/2022

Dátové štruktúry a algoritmy

Zadanie 1

Cvičiaci: MSc. Mirwais Ahmadzai **Author: Michal Darovec**

Čas cvičení: Thursday 11:00 – 12:50 **ID: 116163**

Contents

Úvod

V tejto práci si porovnáme implmentácie rôznych dátových štruktúr. Budeme pracovať s binárnymi stromami a hashovacími tabuľkami. Na každú dátovú štruktúru sú spravené dve rôzne implementácie. Z možností pri binárnych stromoch som si vybral samovyvažovací AVL strom a Splay strom. Pri tabuľkách som si vybral Hash Table Chaining, čiže zreťazovanie jednotlivých prvkov a Open Addressing, presnejšie Linear Probing. Vo všetkých implementáciach sú funkcie na pridanie prvku do štruktúry, vyhľadanie prvku a odstránenie prvku zo štruktúry.

Binárny vyhľadávací strom

Binárny vyhľadávací strom je stromová dátová štruktúra, ktorá má určité vlastnosti. Medzi tieto vlastnosti patrí napríklad to, že ľavý podstrom prvku obsahuje iba prvky s menšou hodnotou ako je hodnota prvku. Pravý podstrom prvku obsahuje iba prvky, ktoré sú od daného prvku väčšie. Ide tu o to, že všetky podstromy sú tiež binárne vyhľadávacie stromy. Keďže tu pracujeme s podstromami, ktoré sú rovnakej štruktúry, budeme veľmi často pracovať s rekurziou.

1. AVL Strom

1.1. Úvod

Pri svojej implementácii AVL stromu som sa snažil robiť ju najjednoduchšie, ako to išlo. AVL strom je typ binárneho vyhľadávacieho stromu, ktorý je samovyvažovací a vyvažuje sa pomocou toho, že rozdiel výšok pravého a ľavého podstromu nikdy nemôže byť väčší ako 1. Toto musí vždy platiť pre všetky prvky v strome.

Na riešenie tohto problému si pomáhame rotáciami, čiže menením pozícií prvkov v strome. Algoritmus rotácie je celkom zrejmý, keď pochopíme princíp rotácie.

```
// left rotation
public NodeAVL leftRotateAB(NodeAVL a) {
    // perform rotation
    NodeAVL b = a.right;
    NodeAVL leftB = b.left;
    b.left = a;
    a.right = leftB;
```

```
// right rotation
public NodeAVL rightRotateAB(NodeAVL a) {
    NodeAVL b = a.left;
    NodeAVL rightB = b.right;
    b.right = a;
    a.left = rightB;
```

Toto vyvažovanie je pri veľkých stromoch veľmi účinné a pri operáciách so stromom ušetríme veľa času. Pri binárnom strome sa nám môže stať, že pri nevhodnom datasete by sme mali časovú komplexitu až O(n). Pri samovyvažovacom strome je však najhorší možný scenár O(log n), čiže pri veľkých číslach často nepredstaviteľný rozdiel.

Ako sme už spomínali, pri binárnych stromoch často používame rekurziu. AVL strom určite nie je výnimkou. Rekurentne prehľadávame strom pri všetkých operáciách so stromom. Na kontrolovanie vyváženosti je samozrejme tiež vhodná rekurzia. Pri každom prvku máme určený parameter: výšku. Keďže vieme, že rozdiel výšok všetkých podstromov nesmie byť väčší ako 1. Preto počítame takzvaný balance factor prvku.

Vypočítame ho presne tak, ako to máme zadefinované:

```
public int calculateBalanceFactor(NodeAVL node) {
    if(node == null) return 0;
    return height(node.left) - height(node.right);
}
```

Pomocou tejto metódy vieme, ako máme vyvažovať strom.

Sú 4 spôsoby ako môžeme dostať nevyvážený strom. Prvý spôsob sa volá tzv. left left case, nastáva vtedy, keď treba spraviť pravú rotáciu lebo strom alebo jeho časť je príliš navážená na ľavú stranu. Druhý spôsob je tzv. right right case, čiže to isté, ale na pravej strane. V tomto prípade treba spraviť pravú rotáciu. Ostatné dva spôsoby sú o niečo komplikovanejšie. Je takzvaný left right case, kde je problém s tým, že je nevyvážený pravý podstrom ľavého podstromu. Následne už môže nastať iba right left case, ktorý má nevyvážený ľavý podstrom pravého podstromu.

```
// left right
if(balance > 1) {
    if(value > root.left.value) {
        root.left = leftRotateAB(root.left);
        return rightRotateAB(root);
    }
}
```

```
// left left
if(balance > 1) {
    if(value < root.left.value) {
        return rightRotateAB(root);
    }
}
```

```
// right right
if(balance < -1) {
    if(value > root.right.value) {
        return leftRotateAB(root);
    }
}
```

```
// left right
if(balance > 1) {
    if(value > root.left.value) {
        root.left = leftRotateAB(root.left);
        return rightRotateAB(root);
    }
}
```

1.2. Insert

Vkladanie do AVL stromu je postavené na rovnakom princípe, ako vkladanie do obyčajného binárneho stromu, avšak s bonusom, že potom vložení prvku treba strom vyvážiť.

Operácia vloženia prebieha takto: Najskôr skontrolujeme, či vkladaný strom nie je null. Ak je, vložíme prvok do stromu ako koreň stromu. Ak nie je, porovnáme hodnotu, ktorú ideme vložiť s hodnotou prvku, ktorý je práve koreň stromu. Ak je náš prvok menší, opakujeme tú istú operáciu na ľavého potomka koreňa. Ak je väčší, opakujeme operáciu vloženia na pravého potomka daného koreňa. Takto sa rekurzívne dostaneme na miesto, kde potrebujeme prvok vložiť a vložíme ho tam.

```
if(root.value > value) {
    root.left = insertToAVL(value, root.left);
} else if(root.value < value) {
    root.right = insertToAVL(value, root.right);
} else if(root.value == value) {
    return root;
}
```

Teraz potrebujeme strom ešte vyvážiť. Proces vyvažovania už sme si opísali vyššie.

1.3. Search

Operácia hľadania je v AVL strome veľmi jednoduchá, stačí postupovať rovnako, ako pri obyčajnom binárnom vyhľadávacom strome. Vždy len rekurzívne zavoláme operáciu v ľavom alebo pravom podstrome, podľa toho, či je naša hodnota vyššia alebo nižšia, ako hodnota aktuálneho prvku.

```
public NodeAVL find(int value) {
    NodeAVL curr = root;
    while(curr != null) {
        if(value == curr.value) return curr;
        if(value > curr.value) curr = curr.right;
        else curr = curr.left;
    }
    return curr;
}
```

1.4. Delete

Delete je najkomplikovanejšia operácia v AVL strome, ako aj v ostatných štruktúrach. Začneme, rovnako, ako pri hľadaní, prezerať strom, kým nenájdeme prvok, ktorý chceme vymazať. Je situácia, že nemá pravého suseda, vtedy by sme jednoducho označili jeho ľavého suseda za koreň, opačne, keby nemal ľavého suseda. Keby mal oboch susedov, musíme nájsť minimálny prvok v jeho pravom podstrome a skopírovať ho na miesto prvku, ktorý ideme vymazať. Následne vymažeme náš prvok. Po vymazaní vyvážime strom.

2. Splay Tree

2.1. Úvod

Splay strom je špeciálny typ samovyvažovacieho stromu. Hlavná myšlienka tohto stromu je to, aby prinášal nedávno vyhľadávané prvky na vrchol stromu – čím bližšie ku koreňu. Keď tým pádom hľadáme posledný vyhľadávaný prvok, nájdeme ho pri časovej komplexite O(1). Keď máme štruktúru, v ktorej sa potrebujeme frekventovane dostať iba ku niektorým prvkom, ostatné sú používané len zriedkavo alebo vôbec, vtedy sa nám oplatí použiť Splay strom.

2.2. Insert

Na to, aby sme mohli pridávať prvky do Splay stromu, potrebujeme najskôr splay funkciu. Táto funkcia je viac opísaná v operácii hľadania, ale v podstate balancuje celý strom a dá posledný hľadaný prvok na pozíciu koreňa. Pri inserte zavoláme túto splay funkciu, ktorá pridá list na pozíciu koreňa, pod ktorý by sme vložili náš nový prvok. Teraz spravíme to, že list, takže koreň, ktorý sme našli, bude pravým potomkom nášho nového prvku, ak bude väčší, ľavým, ak menší ako nový prvok.

2.3. Search

Hľadanie v Splay strome je možné už len zo samotnej operácie splay, ktorú sme spomínali. Pri splayovaní vždy rekurzívne hľadáme hodnotu, ktorú chceme buď vyhľadať pridať k nej alebo odstrániť. Rozdelíme si ju na dve časti, podľa toho, či je hľadaná hodnota menšia alebo väčšia ako koreň. Máme tu 4 scenáre, podobne, ako pri AVL strome. Môže to byť ľavý potomok ľavého potomka, pravý potomok pravého potomka, pravý potomok ľavého alebo ľavý potomok pravého. Všetky tieto scenáre ošetríme dvojitou rotáciou do vhodnej strany. Po splayi bude nájdený prvok vždy koreňom stromu, takže je tým hotový search.

```
// zig zig
// left child of left child
SplayNode leftOfLeft = root.left.left;
if(value < root.left.value) {
    leftOfLeft = splay(value, leftOfLeft);
    root = rightRotate(root); // first rig
}
```

2.4. Delete

Vymazanie prvku v Splay strome je menej náročné ako v AVL. Pomocou splayu hneď premiestnime vymazávaný prvok na miesto koreňa a vymažeme ho. Ostanú nám dva podstromy, ľavý a pravý. Teraz je taktika taká, že nájdeme najväčší prvok z ľavého podstromu a zavoláme naňho splay. Keď bude na vrchu ako koreň ľavého podstromu, určíme si ho ako koreň celého stromu tým, že mu pripíšeme pravý koreň ako pravého potomka.

```
leftTree = splay(value, max(leftTree));
leftTree.right = rightTree; // right ro
return leftTree; // return new root
```

Hashovacie tabuľky

Hashovacie tabuľky sú dátová štruktúra, ktorá mapuje kľúče k hodnotám. Tieto kľúče a hodnoty môžu byť všelijaké nenulové objekty. Na to, aby sme mohli prvky pridávať, vyhľadávať a vymazávať, potrebujeme im priradiť hash, ktorý dosiahneme funkciou hashCode().

3. Chaining

3.1. Úvod

Hashovacie tabuľky obsahujú páry kľúčov a hodnôt. Tieto kľúče sú vždy iné, špeciálne. Lenže, keď máme hashovaciu funkciu a nemáme dostatok prvkov v tabuľke, tak musíme nejako vyriešiť, kam dáme prvky, ktoré majú svoje miesta už obsadené. Táto situácia sa volá kolízia a dá sa riešiť rôznymi spôsobmi. Najskôr ju budeme riešiť pomocou reťazenia – chainingu prvkov na danom indexe.

Zvolil som si takúto hashovaciu funckiu:

```
public int hashCode(int key) {
    return (int) Math.floor(elements.size() * (key * (0.3 % 1)));
}
```

3.2. Insert

Pridávanie prvkov do hashovacej tabuľky vie spôsobovať kolízie, preto si musíme dávať pozor, ako ich ošetriť. Teraz ideme použiť chaining. Budeme vytvárať spájané zoznamy prvkov na danom indexe. Vždy, keď nájdeme ďalší prvok, ktorý by mal ísť na istý index, ktorý nebude voľný, pripojí sa do zoznamu prvkov, ktoré index už obsahuje. Jeden prvok bude takto nadväzovať na druhý a vždy bude miesto pre ďalšie prvky do zoznamu.

```
// add new element to the start of the chain
NodeHashChain newElement = new NodeHashChain(key, value, hash);
elements.set(index, newElement);
newElement.next = elementAtIndex;
realSize++;
```

Keď pridávame prvok, je najľahšie ho pridať na začiatok zoznamu prvkov na indexe. Prvok, ktorý bol dovtedy prvý na indexe, iba posunieme na druhé miesto pomocou odkazu prvok.next.

3.3. Search

Hľadanie v tabuľke nie je náročné, podobne ako pri AVL strome. Treba si iba nájsť správny index v tabuľke a keď už máme index, tak nám stačí prejsť spájaný zoznam prvkov, ktoré sa nachádzajú na danom indexe. Hodnotu nájdeného prvku vrátime.

```
while(elementAtIndex != null) {
    if(elementAtIndex.hash == hash && elementAtIndex.key == key) {
        return elementAtIndex.value;
    }
    elementAtIndex = elementAtIndex.next;
}
```

3.4. Delete

Pri vymazávaní prvku si najskôr nájdeme index prvku, ako pri hľadaní a prehľadáme daný index. Ak je vymazávaný prvok na začiatku zoznamu, len dáme na začiatok druhý a na prvý sa zabudne. Ak nie je na začiatku, musíme si strážiť predchádzajúci prvok cez premennú. V tomto prípade ako keby preskočíme vymazávaný prvok tým, že žiadny prvok ho nebude mať v odkaze prvok.next, predchádzajúci.next už bude ukazovať na ďalší.

```
else if(elementAtIndex.hash == hash &&
  previous.next = elementAtIndex.next;
  realSize--;
  return elementAtIndex.value;
```

4. Open Addressing

4.1. Úvod

Hashovacie tabuľky sa dajú robiť viacerými spôsobmi. Kolízie treba ošetriť vždy a existuje viacero spôsobov, ako na to. Teraz ich budeme riešiť cez otvorené adresovanie. Ide jednoducho o to, že ak narazíme na plné miesto, tak budeme chodiť po všetkých ďalších indexoch a budeme sa pozerať, kam by sa daná hodnota dala uložiť.

Teraz som si zvolil inú, jednoduchšiu hashovaciu funkciu:

```
public int hashCode(int key) {
    return key % elements.size();
}
```

4.2. Insert

Pridávanie prvku do tejto tabuľky bude pomerne ľahké, keď sme na indexe, tak iba sledujeme, či nie je voľné miesto a posúvame sa ďalej vždy o jedno miesto, ktoré následne zmodulujeme, aby sme nedostali väčšie číslo, ako veľkosť tabuľky. Vkladať môžeme aj na miesta, kde sú vymazané prvky.

```
// if found free space or deleted element -> insert
if(elements.get(index) == null || elements.get(index).key == -1) {
    NodeHashLinear newNode = new NodeHashLinear(key, value);
    elements.set(index, newNode);
    size++;
```

4.3. Search

Hľadanie prvku v tabuľke funguje na podobnom princípe, ako vkladanie a vymazávanie, minimálne do bodu nájdenia indexu prvku. Index stále zväčšujeme o 1 a zakaždým ho zmodulujeme veľkosťou tabuľky. Ak nájdeme kľúč, vrátime jeho hodnotu. Keď nájdeme prvok, tak musíme skontrolovať, že to nie je vymazaný prvok, ktorému dávame kľúče a hodnoty -1.

```
while(elements.get(index) != null && elements.get(index).key != -1)
    if(i > elements.size()) return null;
    if(elements.get(index).key == key) {
        return elements.get(index).value;
    }
```

4.4. Delete

Vymazanie prvku je znovu náročnejšie na implementáciu. Najskôr nájdeme prvok, tak isto, ako v search operácii. Ak prvok nenájdeme, môžeme rovno ukončiť funkciu. Keď nájdeme index na vymazanie, daný prvok si označíme, dáme mu -1 ako kľúč a null ako hodnotu. Zmenšíme aj veľkosť existujúcich prvkov v tabuľke. Všetky prvky, ktoré sa nachádzajú v tabuľke za vymazávaným prvkom sa vymažú a následne sa znovu pridajú cez funkciu insert.

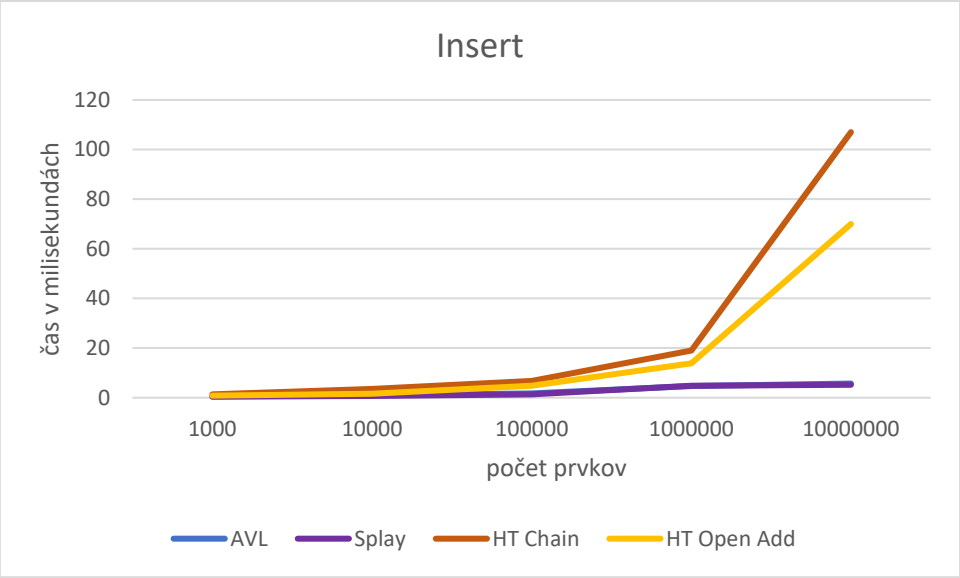
```
NodeHashLinear nodeToDelete = elements.get(index);
nodeToDelete.key = -1;
nodeToDelete.value = null;
existingSize--;

// re-add other affected elements
index = (index + 1) % elements.size();
while(elements.get(index) != null) {
    int tempKey = elements.get(index).key;
    String tempValue = elements.get(index).value;
    elements.get(index).key = -1;
    elements.get(index).value = null;
    addElement(tempKey, tempValue);
    existingSize--;
    index = (index + 1) % elements.size();
}
return nodeToDelete;
```

Testovanie

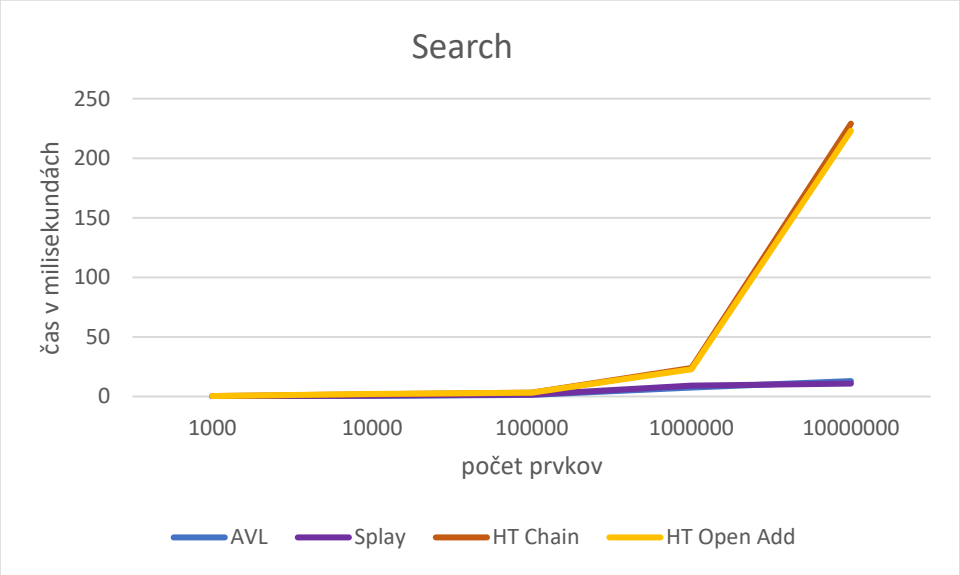
Pri vkladaní som testoval z datasetu náhodných čísel v poli rýchlosť všetkých štyroch implementácií. Náhodné čísla boli vytvorené naplnením poľa a následným premiešaním čísel v poli. Čas bol počítaný v nanosekundách a do tabuľky následne prepísaný v milisekundách. Testované boli veľkosti od 1000 prvkov až po 10 miliónov prvkov.

1. Insert



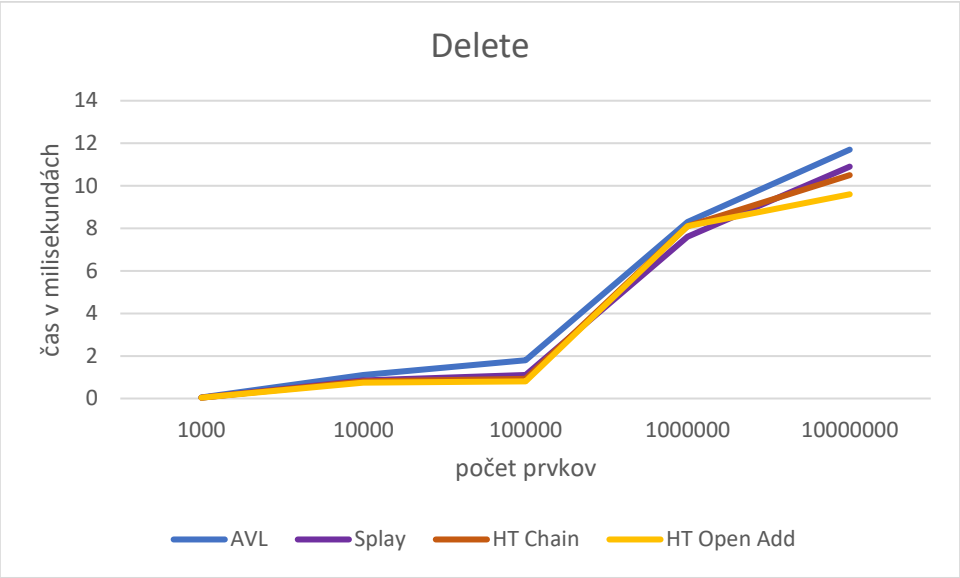
Stromy boli pozoruhodne rýchle v porovnaní s tabuľkami. Ťažšie na implementáciu, ale rýchlejšie a efektívnejšie.

2. Search



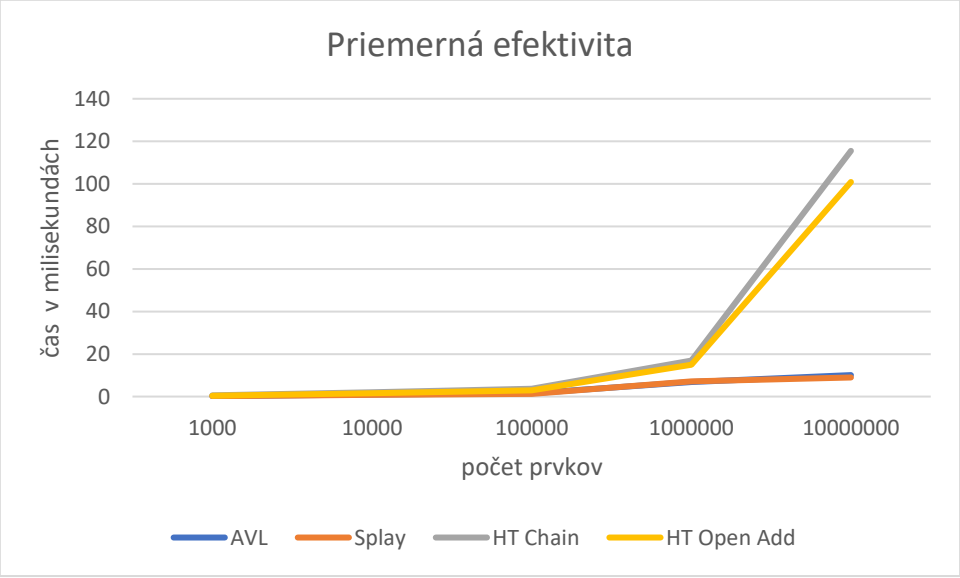
Hľadanie bolo celkovo veľmi podobné vkladaniu. Vo všetkých prípadoch to dopadlo podobne, jediný menší rozdiel je v tabuľkách. V tabuľke s otvoreným adresovaním sa hľadá rýchlejšie, zatiaľ čo pridávanie bolo pomalšie.

3. Delete



Pri operácii delete si môžeme všimnúť vyrovnanie rýchlostí všetkých dátových štruktúr. Najpomalšie zvláda vymazávanie AVL strom, ale rozdiel je minimálny. Pri poslednom testovaní bolo dokonca najrýchlejšie otvorené adresovanie. Je to z dôvodu, že prístup k dátam je jednoduchší, ako k dátam v stromoch a nepotrebujeme prechádzať jednotlivé spájané zoznamy v chainingu.

4. Priemer výsledkov



Následne môžete vidieť náhľad spriemerovania časov všetkých operácií pre kompletné porovnanie týchto dátových štruktúr.

Zdroje

https://www.geeksforgeeks.org/

https://www.programiz.com/

https://www.javatpoint.com/