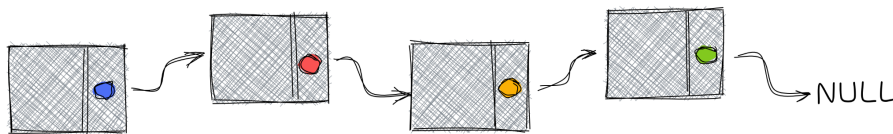


Data Structures and Algorithms

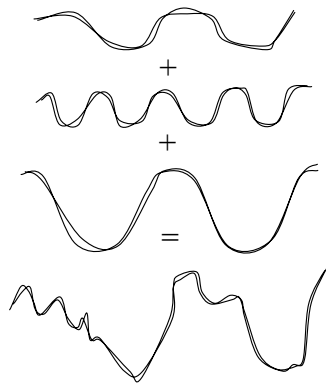
Supporting Material



Hyrje

E fillojmë këtë material me një histori të shkurtër:

*Dekompozimi i një funksioni në komponenta me frekuenca të ndryshme (Fig.1) është një operacion i rëndësishëm për funksionimin e shumicës së gjërave në internet. Ky dekompozim bëhet përmes llogaritjes së Discrete Fourier Transform (DFT) të një sekuence vlerash. Sipas definicionit, ky kalkulim ka kompleksitet $O(n^2)$. Me zbulimin e algoritmit **Fast Fourier Transform (FFT)** për llogaritje, kompleksiteti është zvogëluar në $O(n \cdot \log n)$. Nëse një llogaritje me algoritmin e kaluar do të merrte disa vite, algoritmi i ri mund ta përfundojë për vetëm disa ditë!*



Strukturat e të Dhënave dhe Algoritmet janë dy prej fushave më të rëndësishme, e njëkohësisht më komplekset, për funksionimin e të gjitha aspekteve të teknologjisë informative. Një njohje e mirë e këtyre temave do të rrisë drastikisht efikasitetin e kodit tuaj e do të ju ndihmojë në shkruarjen e kodit më të pastër, të lexueshëm, dhe të shpejtë.

Një diskutim i gjerë rreth këtyre temave normalisht nuk do të mund të përmbahej brenda kornizave të këtij materiali. Për këtë arsye, ky material duhet të përdoret si lexim shtesë teorik krahas trajnimit praktik në të cilin keni marrë pjesë. Përmbajtja e materialit është si më poshtë:

- Hyrje
- Si funksionon kompjuteri
- Të dhënat Primitive
- Vargjet (Dinamike)
- Kompleksiteti i Algoritmeve
- Stacks
- Listat e Lidhura
- Hash-Funksionet dhe Hash-Tabelat
- Për lexim të mëtejshëm

Shpresojmë që ky material, krahas trajnimit, të ju shërbejë për marrjen e njohurive të reja dhe për nxitjen e kuriozitetit rreth Strukturave të të Dhënave dhe Algoritmeve!

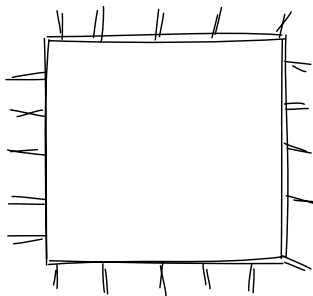
Si funksionon Kompjuteri?

Nëse i pyesim disa persona se si funksionon kompjuteri, do të merrnim përgjigje të ndryshme në nivele të ndryshme të kompleksitetit: Dikush do të jepte një përgjigje lidhur me monitorin, dikush me *Hardware dhe Software*, dikush do fliste për Sistemin Operativ, dikush për Procesorin dhe RAM-in, dikush ndoshta për qarqe logjike, e ndoshta dikush për transistorë, elektricitet, e ligje të fizikës. Një përgjigje precize ndaj pyetjes "*Si funksionon kompjuteri*" prandaj është shumë e vështirë për t'u konstruktuar.

Ne nuk do e analizojmë kompjuterin në një nivel të lartë si Sistemi Operativ e as në një nivel të ulët si transistorët. Ne do e analizojmë atë në një nivel mesatar të kompleksitetit ku do flasim për Procesorin dhe Memorien Kryesore. Kjo qasje na bën të thjeshtë më vonë kuptimin e Strukturave të të Dhënave dhe funksionimin e Algoritmeve.

Procesori (CPU)

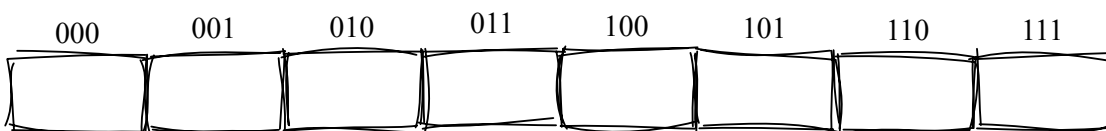
Shpesh procesori quhet *Truri i kompjuterit* mirëpo kjo nuk është komplet e vërtetë. Truri jonë mund të mendojë dhe të ruajë informacion njëkohësisht – procesori dallon në këtë aspekt. Procesori mund të pranojë instruksione si pulse elektrike që reprezentojnë vlera binare (shembull: 00110011) dhe të bëjë dicka kur e pranon atë instruksion. Këto instruksione mund të jenë rreth lëvizjes së të dhënave, operacioneve aritmetike ose logjike, etj.



Memoria Kryesore (RAM)

Memoria Kryesore është pjesa tjetër e *trurit* të kompjuterit. Këtu ruhen të dhënat dhe instruksionet me të cilat Procesori punon aktivisht. Normalisht, këto të dhëna janë të ruajtura në formë binare, zakonisht të ndara në bytes (8 bit).

Një vecori interesante e RAM-it është se ajo ka një aspekt linear. Pamja e saj e thjeshtë konceptuale është si më poshtë:



Vërejmë se ajo është e përbërë nga disa qeliza të *renditura*, ku secila e ka nga një **adresë memorike** unike që tregon lokacionin e qelizës. Secila qelizë do të supozojmë se përmban 1byte.

RAM-i e ka një veti shumë interesante: ajo mund të kontrollojë vlerën që është ruajtur në një qelizë në kohë konstante, pavarësisht vendndodhjes së qelizës, duke pasur vetëm adresën e qelizës si informacion. Për këtë

arsye ajo quhet *Random Access Memory*, pra mund t'i qaset memories në mënyrë të cfarëdoshme. Kjo dallon prej medimeve të tjera të ruajtjes së të dhënave ku të dhënat duhet të gjeihen në mënyrë sekuenciale: mediumet magnetike ose optike.

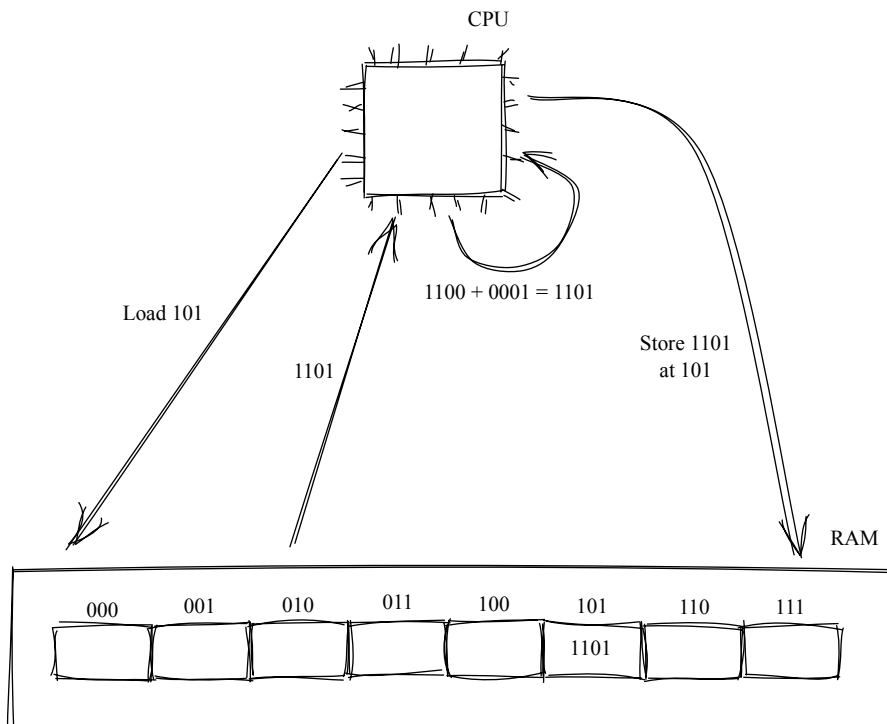
Komunikimi mes Procesorit dhe RAM-it

Procesori e shfrytëzon RAM-in si vend për të ruajtur dhe për t'ju qasur të dhënave përkohësisht. Konsiderojmë rastin e mëposhtëm:

Kemi numrin `1100` (numër decimal 12) të ruajtur në RAM në adresën `101`. Detyrë e procesorit është që atë vlerë ta rrisë për 1. Si do të rrjedhte komunikimi mes Procesorit dhe RAM-it për të arritur këtë?

Ne e dimë se rezultati është 13, mirëpo procesori nuk mund ta rritë vlerën direkt. Ai duhet të ekzekutojë këta hapa:

- Merre vlerën në adresën `101`
- Llogarit shumën e numrit të marrë me `0001`
- Ruaje rezultatin e kalkulimit në adresën `101`



Ky komunikim është i vazhdueshëm dhe Procesori merr edhe instruksionet edhe të dhënat nga RAM-i. Kjo është çfarë ndodh në prapaskenë kur ne shkruajmë kod të stilit si më poshtë:

```
x = 12
x++
```

Të dhënat Primitive

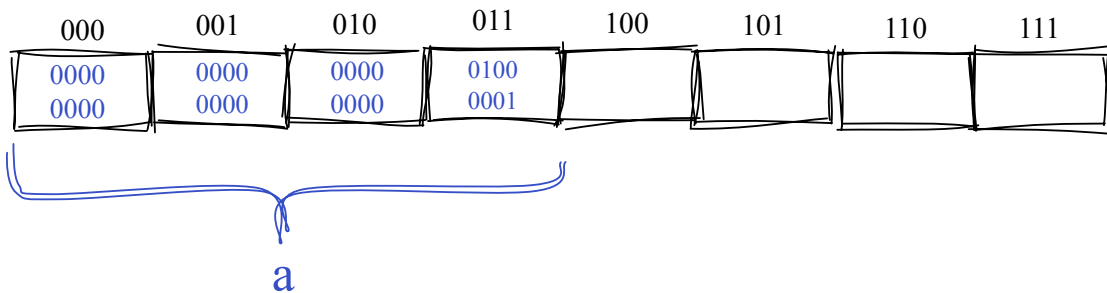
Të dhënat primitive janë forma më e thjeshtë e ruajtjes së të dhënave. Ato kanë madhësi të fiksuar dhe ruhen në një numër të vogël bajtësh. Gjuhët e larta programuese në një formë apo tjetrën lejojnë të punohet me vlera primitive dhe të manipulohet me to.

Një vlerë primitive ka një madhësi të definuar, le të themi 1 byte. Kjo vlerë mund të ruhet pa problem në një qelizë të memories kryesore. E pamë se procesori mund të ketë qasje në këtë vlerë. Në gjuhën programuese Java, mund të krijojmë një vlerë primitive si më poshte:

```
int a = 65;
```

Në këtë rast kemi krijuar një vlerë primitive të tipit `int` dhe në të kemi ruajtur vlerën e numrit `65`.

Madhësia e tipit `int`, sipas definicionit në Java, është 4byte. Në memorien kryesore kjo vlerë do dukej kështu:



Shohim se në memorie gjithcka ruhet si vlerë binare. Në rastin tonë konvertimi bëhet përmes standardit Two's Complement i cili na lejon të reprezentojmë edhe numra negativ, mirëpo tipet e tjera të të dhënave si shkronjat dhe numrat me presje konvertohen në numra binarë sipas standardeve të ndryshme.

Në Java kemi edhe tipe të tjera primitive me madhësi të ndryshme:


```
long x = 34535L;  
char a = 'A';  
//...
```

Vlerat `long` kanë madhësi 8byte kurse vlerat `char` 4byte. Kur një vlerë është më e madhe se një qelizë, atëherë ajo zgjatet për të nxënë edhe qelizat fqinje, sic u pa më lartë me vlerën `int`.

Operacionet me vlera primitive janë të thjeshta në gjuhët e larta programuse. Nëse dëshirojmë të mbeldhim dy numra dhe ta ruajmë rezultatin në një variablë të tretë, nuk ka nevojë të shkruajmë detajisht se cka duhet të ndodh mes procesorit dhe memories, por thjeshtë e reprezentojmë logjikën:

```
int a = 64;  
int b = 13;  
int c = a + b;
```

Kjo vlen gjithmonë duhe respektuar rregullën e artë se *rezultati i anës së djathtë ruhet në anën e majtë*:

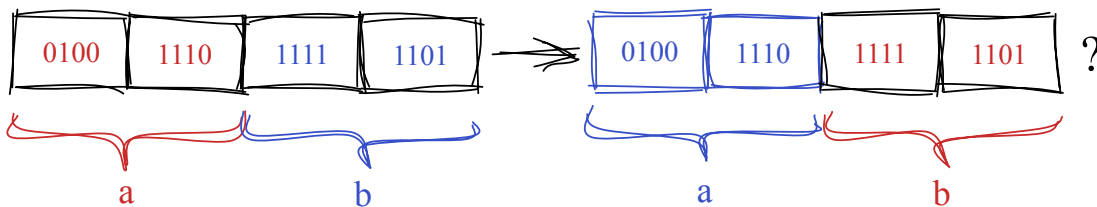

`int c = a + b;`

Në këtë formë mund të ruajmë çfarëdo të dhëna të thjeshta dhe të manipulojmë me to.

Nuk ka shumë algoritme të cilat mund t'i diskutojmë rreth vlerave primitive, për arsyen se janë shumë të thjeshta. Prapsepapë, të shohim një algoritëm të lezetshëm.

Swap

Një rast i shpeshtë është që të ndërrojmë vendet e dy vlerave. Nënkuptojmë që vlera e variablës `a` të ruhet në `b`, dhe vlera e variablës `b` të ruhet brenda `a`. Vizualisht, kemi problemin e mëposhtëm:

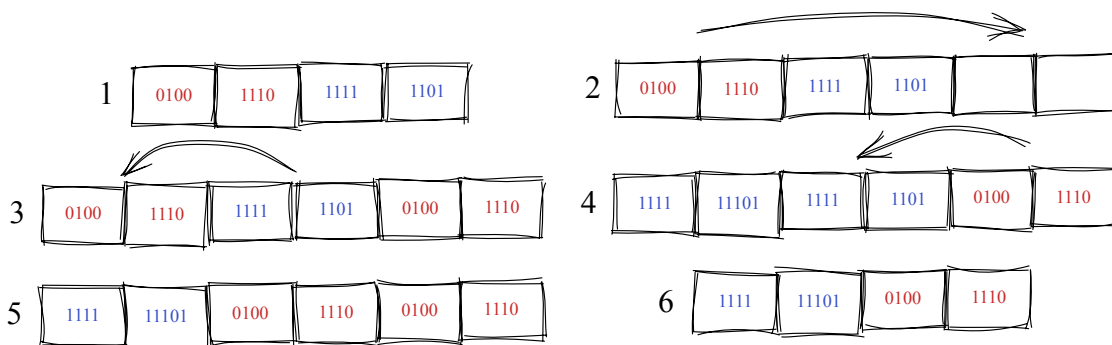


Si duhet t'i qasemi këtij problemi? Mund të mendojmë se problemin mund ta zgjidhim si më poshtë:

```
a = b; //?  
b = a; //?
```

Mirëpo kjo do të rezultonte në gabim sipas rregullës së artë. Kjo është për shkak se në rreshtin e parë vlera e `b` ruhet brenda `a`, gjë që rezulton në humbjen complete të vlerës së `a` !!!

Kjo na jep të mendojmë se vlerën e `a` duhet ta ruajmë diku tjetër deri sa të bëhet zëvendësimi. Me këtë ide, e krijojmë algoritmin si në vijim, ku variabla e kuqe ndërrohet me atë të kaltrën:



Tani thjeshtë mund ta shkruajmë algoritmin në Java:

```
int a = 5;
int b = 6;

int c;
c = a;
a = b;
b = c;
```

Ky algoritëm ishte prej më të thjeshtit e mundshëm por prapseprapë na shtyn të mendojmë zgjidhje interesante për problemet që na shfaqen.

Një vecori interesante është se shumica e algoritmeve janë të implementuara, dhe thjesht duhet të mësohemi t'i përdorim si duhet. Në rastin e ndërrimit të vendeve, në Java ne e bëmë zgjidhjen më të mirë të mundshme mirëpo në JavaScript do mjaftonte e mëposhtmjia:

```
let a = 5;
let b = 6;

[a, b] = [b, a];
```

Edhe pse është më e thjeshtë për t'u shkruar, në prapaskenë JavaScript e përdor algoritmin e njëjës me atë që e shkruam për Java për të bërë ndërrimin e vendeve.

Vargjet (Dinamike)

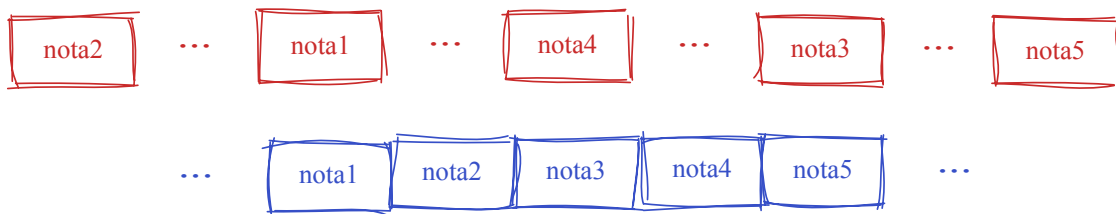
Vlerat Primitive janë të thjeshta por jo edhe shumë të përdorshme kur kemi të bëjmë me numër më të madh të të dhënave. Në këtë seksion do të shikojmë një mënyrë të re për të ruajtur të dhënat në formë të listës. Supozojmë se duhet të punojmë me disa të dhëna, për shembull notat e Benit për 5 lëndët e semestrit të parë. Me aq sa kemi përmedur deri tash, këto nota mund ti ruajmë në këtë formë si vlera primitive:

```
int nota1 = 9;
int nota2 = 10;
int nota3 = 10;
int nota4 = 8;
int nota5 = 8;
```

Kjo duket si zgjidhje e mirë, por kur duhet të punojmë me këto nota, apo kur dëshirojmë të shtojmë lëndë të reja, apo kur duam të përfshijmë të dhëna për një numër të madh studentësh, gjërat fillojnë të komplikohen. Kjo është për arsye se nuk ka ndonjë lidhje logjike mes notave *në kod* edhe pse ka lidhje logjike mes tyre *në jetën reale*.

Vargu Statik

Një zgjidhje elegante për të punuar me këso të dhënash është **Vargu (Array)**. Ideja është që këto variabla të mos ruhen në vende të ndryshme në memorie, por të ruhen së bashku në qelizë fqinje:



Fakti që vlerat ruhen së bashku jo vetëm tregon një lidhje logjike mes tyre por na jep edhe përfitime të tjera, sic do të shohim më tej. Në gjuhën programuese Java, vargu definohet thjeshtë si më poshtë:

```
int[] notat = new int[5];
```

Një veti e rëndësishme për vargjet, sikur te vlerat primitive, është alokimi i memories për ruajtjen e të dhënave. Me këtë nënkuptojmë se kur deklarojmë një variabël `int a;` automatikisht alokohen 4byte në memorie për ruajtjen e asaj vlere. Kjo është me rëndësi sepse, sic e kemi përmendur, memoria është lineare dhe qelizat para dhe prapa variablës alokohen për qëllime të tjera.

Ngjashëm në vargje, *madhësia e vargut duhet të përcaktohet paraprakisht* që të alokohet memorie e mjaftueshme. Në shembullin më lart, numri `5` tregon numrin e elementeve që do t'i ruajmë brenda vargut. Njëkohësisht, edhe tipi i vargut është i përcaktuar që të dihet saktë numri i bajtëve që duhet të alokohen. Në rastin tonë vlera `int` ka 4byte, andaj alokohen $5 * 4 = 20\text{byte}$.

Memoria u alokua, tani vjen pyetja se si t'i qasemi elementeve? Një koncept interesante është koncepti i **indeksit**. Elementeve i qasemi me anë të pozitës së tyre në varg. Indekset në shumicën e gjuhëve programuese fillojnë nga zero, duke përfshirë gjuhën Java, mirëpo disa, shembull gjuha *Lua*, fillon numërimin nga numri një. Tani, mund të caktojmë notat si më poshtë:

```
notat[0] = 9;
notat[1] = 10;
notat[2] = 10;
notat[3] = 8;
notat[4] = 8;
```

Tash lehtë mund të punojmë me vlerat dhe të iterojmë në to me funksionalitete të ndryshme të gjuhëve - shembull me një for-loop:

```
for(int indeksi = 0; indeksi < notat.length; indeksi++){
    doSomething(notat[i]);
}
```

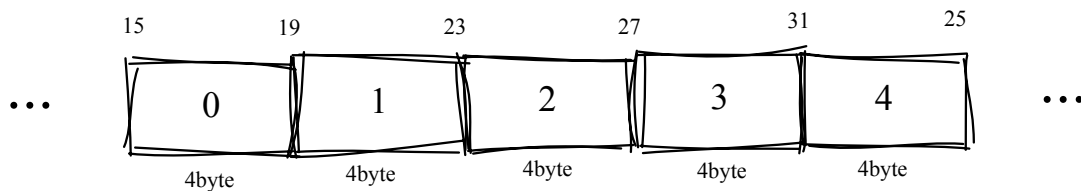
Nëse nuk ju ka shkuar në mend, si mund ta gjejë një varg përmes indeksit se ku gjendet variabla? Pse të ruajmë vlerat së bashku e jo ndamas? Cka përfitojmë nga kjo?

Address Polynomial

Vargu mund të gjejë pozitën e variablës përmes indeksit shumë thjeshtë duke përdorur **address polynomial**. Një kuptim i thjeshtë i saj është ky:

Ne e kemi adresën se ku fillon vargu, të themi qeliza `15`. Gjithashtu, e dimë edhe tipin e vargut, `int`, dhe madhësinë e kësaj vlere primitive `4byte`. Tani, nëse duam të gjejmë variablën me pozitën e tretë, mjafton

të llogarisim $15 + 3 \cdot 4 = 27$. Pra vlera në pozitën 3 fillon nga qeliza 27. Në këtë formë e gjejmë adresën e saktë dhe i qasemi asaj menjëherë në kohë konstante.



Operacionet me Vargje Statike

Sa herë kemi të bëjmë me struktura të të dhënave, duhet të mendojmë se si i qasemi elementeve, si e ndryshojmë një element, si e shtojmë një element të ri, dhe si e fshijmë një element. Gjithashtu duhet të mendojmë edhe për kohën që na merr t'i qasemi një elementi.

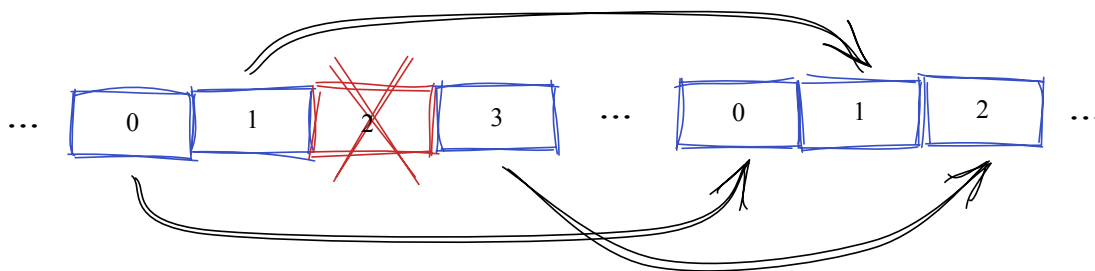
Qasja dhe Ndryshimi i elementeve

E diskutuam edhe më herët se përmes *address polynomial* i qasemi një elementi në kohë konstante. Kjo do të thotë se *pavarësisht numrit të elementeve, mjafton një kalkulim që të gjejmë adresën e elementit me indeks të caktuar*. Kjo mund të duket triviale mirëpo nëse kemi një varg me 10000 elemente prapë nuk ka ndryshim në shpejtësi. E sic do të shohim më tutje, jo gjithmonë qasja e elementit është konstante! Për më shumë, pasi qasja është konstante atëherë edhe ndryshimi qartë është konstant.

Fshirja dhe Shtimi i elementeve

Deri tani gjithcka ishte perfekte me vargjet, por tani gjërat fillojnë të komplikohen. Si ta fshijmë një element nga vargu?

Një element nuk mund të fshihet trivialisht nga vargu. Pasi vargu nxen një bllok të ngjitur të memories, edhe pas fshirjes ne duhet të mbajmë bllokun e ngjitur. Mënyra e vetme për të bërë këtë është që të krijojmë një varg të ri i cili përmban të gjitha elementet **përveq elementit të cilin dëshirojmë ta fshijmë*. Pra duhet të krijojmë një varg me madhësi për 1 më të vogël. Vizualisht, ky proces mund të duket kështu:



Në figurë bëmë fshirjen e elementit në pozitën 2 nga një varg me 4 elemente. Në Java, një gjë të tillë do e bënim në formën:

```
int[] array = new int[4];
int deleteElementIndex = 2;
```

```
int[] newArray = new int[array.length - 1];

for(int i = 0; i < deleteElementIndex; i++){
    newArray[i] = array[i];
}

for(int i = deleteElementIndex + 1; i < array.length; i++){
    newArray[i - 1] = array[i];
}
```

Vërejmë se fshirja *nuk është konstante* sepse për një varg me 5 elemente duhet të bëjmë 4 bartje, kurse për një varg me 10000 elemente duhet të bëjmë 9999 bartje. Kur numri i veprimeve varet linearisht nga numri i elementeve këtë e quajmë *kompleksitet linear* apo **O(n)** ku *n* është numri i elementeve në varg. Nga ana tjetër, nëse dicka bëhet në kohë konstante atë e shënojmë **O(1)**. Ky notacion quhet *Big O Notation* për të cilin do flasim më vonë.

Të shtojmë një element ka të njejtat probleme. Si të shtojmë elementin e 5-të në një varg me gjatësi 4? Përgjigjja është e njejtë: Krijojmë një varg të ri me `N+1` elemente dhe i bartim të gjitha elementet e vargut të vjetër në vargun e ri - dhe në fund e shtojmë elementin e ri. Proveni të shënoni algoritmin në Java për këtë funksionalitet.

Vargjet Dinamike

E pamë se me vargje statike mund të merremi me lista të vlerave shumë thjeshtë...mirëpo ato nuk janë edhe pa mangësi. Mangësia e parë është se *gjatësia e tyre është e fiksuar* dhe duhet të dihet në momentin e inicializimit. Por cka nëse nuk e dimë këtë gjatësi? Apo për më tepër, cka nëse na duhet të shtojmë elemente të reja më vonë pas inicializimit apo të fshijmë elemente?

Fatkeqisht, nuk ekziston një zgjidhje perfekte që na zgjidh të gjitha këto probleme. Mirëpo lehtë mund t'i afrohem asaj. Përgjigja është tek **Vargjet Dinamike**.

Vargjet Dinamike ofrojnë një abstraksion i cili në prapaskenë shfrytëzon një Varg Statik por na ekspozon funksione me të cilat na plotësojnë të gjitha nevojat tona. Një spjegim i thjeshtë në kod është ky:

```
//për thjeshtësi, supozojmë se Vargu mban vetë vlera të tipit int
class DynamicArray{
    private int[] arr;

    public DynamicArray(int size){
        this.arr = new int[size];
    }

    public int get(int index){
        return this.arr[index];
    }

    public void set(int index, int value){
        this.arr[index] = value;
    }

    public void add(int value){
        //kodi për shtimin e elementit të ri
    }
}
```

```

public int delete(int index){
    //kodi per fshirjen e elementit
}
}

```

Kodi më lartë tregon thjesht idenë e abstraksionit. Vërejmë se i kemi funksionet për të gjitha operacionet dhe si janë implementuar ato nuk është me rëndësi për përdoruesin e klasës. Por si do ta implementonim vetë këtë funksionalitet?

Për kthimin apo ndryshimin e një elementi thjesht e shfrytëzojmë vargun e brendshëm statik, me kompleksitet $O(1)$. Për fshirjen e një elementi mund të përdorim algoritmin e spjeguar në shembullin e kaluar, me kompleksitet $O(n)$. Por cka nëse dëshirojmë të shtojmë një element ekstra në një indeks më të madh se gjatësia e vargut?

Zgjidhje triviale për këtë të fundit është që të rrisim gjatësinë e vargut, duke e kopjuar atë në një varg tjetër, ashtu që elementi i shtuar është në fund të vargut. Në këtë rast kemi kompleksitet $O(n)$ për cdo shtim të ri në varg. Kjo nuk është fare efikase kur kemi të bëjmë me një numër të madh të të dhënave dhe shtojmë të dhëna të reja në varg në mënyrë konstante. Andaj na duhet një zgjidhje tjetër.

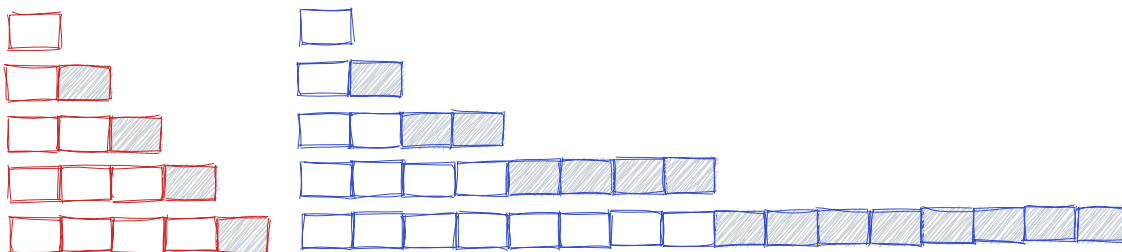
Faktori i zmadhimit

Që të zvogëlojmë kompleksitetin e shtimit të elementeve të reja do të shohim një zgjidhje shumë interesante: kur shtojmë element të ri, gjatësinë nuk e rrisim vetëm për 1. Ideja është që gjatësinë ta rrisim më shumë sesa 1 kur e shtojmë një element të ri.

Ta mendojmë kështu: nëse gjatësia e vargut është N , me të gjitha elementet janë të mbushura, dhe ne dëshirojmë të shtojmë element të ri, vargun e ri që e krijojmë nuk e inicializojmë me gjatësi $N+1$ por me gjatësi $k*N$, ku k është një numër pozitiv më i madh se 1 (**growth factor**). Pse?

Më herët, cdo shtim në varg merrte kohë $O(n)$ sepse të gjitha elementet duhet të barteshin. Tani, vetëm nganjëherë ndodh që kemi kohë $O(n)$ dhe pas kësaj për shumë elemente të tjera kemi kohë $O(1)$ pasi kemi lënë vende rezervë të thata për elemente të reja. Vendet e thata janë prezente vetëm në brendësi të strukturës së të dhënave dhe nuk mund të modifikohet nga jashtë. Kjo na lejon që shumicën e kohës të kemi kompleksitet $O(1)$, gjë që quhet *amortized constant time*.

Si të zgjedhim faktorin e zmadhimit? Nuk ka një përgjigje të saktë për këtë mirëpo mund ti shikojmë disa raste. Në gjuhën Java, klasa `ArrayList` është një implementim i vargut dinamik e cila shfrytëzon faktorin e zmadhimit 2 - pra e dyfishon gjatësinë kur mbushet vargu. Në Python faktori i zmadhimit është 1.25, kurse në gjuhën Go varion nga 1.25 deri në 2. Më poshtë shohim edhe një vizualizim të këtij procesi:



Në rastin e parë për cdo element të shtuar e rrisim gjatësinë për 1 (duke rezultuar në $O(n)$ për cdo insertim) kurse në rastin e dytë e rrisim gjatësinë me faktor 2 duke rrujtur vende rezervë. Edhe pse zgjidhja e dytë konsumon më shumë memorie të pashfrytëzuar, prapë ja vlen në aspekt të kohës.

Një modelim i tillë që e bëmë në këtë seksion, tregon një shembull të një *Strukture të dhënash*: ofrojmë zgjidhje për probleme të caktuara duke ekspozuar funksione që i manipulojnë të dhënat duke e fshehur kompleksitetin e implementimit. Kjo e lë përdoruesin që të mendojë vetëm në aspektin logjik dhe jo atë të implementimit. Për një shfrytëzues i cili dëshiron të shtojë të dhëna në një listë, nuk i intereson se sa është faktori i zmadhimit apo si funksionon memoria - mjafton të ketë qasje në funksionet për të shikuar, ndryshuar, shtuar, dhe fshirë elementet!

Vlen gjithashtu të përmendet se ka implementime të ndryshme për një varg dinamik të cilat nuk e përcjellin ecurinë që përshkruam këtu. Proveni të bëni një implementim të thjeshtë të një vargu dinamik në gjuhën tuaj të preferuar programuese!

Algoritme me vargje

Kemi algoritme të ndryshme në vargje. Algoritme të thjeshta ishin ato për fshirjen apo insertimin e elementeve. Algoritme të njohura, bashkë me kompleksitetet e tyre:

- Gjetja e elementit më të madh - **$O(n)$**
- Reverse Array - **$O(n)$**
- Gjetja e elementit të dytë më të vogël - **$O(n)$**
- Renditja e elementeve nga më i vogli tek më i madhi - **$O(n \log n)$**

Algoritme interesante për të gjitha këto mund të gjeni online dhe mund t'i provoni vetë. Në rastin e renditjes, algoritmet më të thjeshta funksionojnë në **$O(n^2)$** mirëpo ekzistojnë edhe algoritme shumë më të shpejta në **$O(n \log n)$** . Shohim algoritmin më të thjeshtë nga ata më lart.

Reverse Array

Algoritmi më i thjeshtë do ishte të krijojmë një varg të ri me gjatësi të njejtë dhe të kopjojmë duke filluar nga fundi secilin element nga vargu fillestar, si më poshtë:

```
int[] array = new int[5];
int[] reverse = new int[array.length];

for(int i = array.length - 1; i >= 0; i--){
    reverse[array.length - i - 1] = array[i];
}
```

Edhe pse kjo ka kompleksitetet të kohës **$O(n)$** , ajo ka edhe kompleksitet të *hapsirës* **$O(n)$** për shkak se krijojmë një varg të ri me gjatësi të njejtë. Si mund ta përmirsojmë këtë?

Nëse na kujtohet algoritmi **Swap** nga më herët, mund ta shfrytëzojmë për të ndërruar vendet: i fundit me të parin, i parafundit me të dytin, e kështu me rradhë:

```
int array = new int[5];

for(int i = 0; i < Math.ceil(array.length / 2) - 1; i++){
    swap(array[i], array[array.length - i - 1]);
}
```

Tani kompleksiteti i kohës është **$O(n)$** mirëpo nuk shfrytëzojmë fare memorie ekstra.

Kompleksiteti i Algoritmeve

Përgjatë tekstit do të vëreni shprehje si $O(n)$, $O(n \cdot \log n)$, etj. Në këtë seksion do të përshkruajmë jo-formalisht kuptimin e këtyre shprehjeve.

Algoritmet na nevojiten për kryerjen e punëve dhe sa më i shpejtë algoritmi aq më mirë. Mirëpo si t'i krahasojmë algoritmet? Krahasimi përmes kohës së ekzekutimit, cikleve të procesorit, etj, të gjitha rezultojnë në të pasakta. Me anën e *Big O Notation* do të masim rigorozisht dallimin në shpejtësi mes algoritmeve.

Secili algoritëm punon me një numër të dhënash të cilën e merr si hyrje. Një algoritëm që rendit numrat merr një listë me gjatësi n si hyrje, kurse një algoritëm që analizon zërin merr n vlera me matje të ndryshme të zërit përgjatë kohës. Prandaj, kur krahasojmë dy algoritme, ato duhet ti krahasojmë për të gjitha vlerat e n .

Analizojmë funksionin që iteron nëpër të gjitha elementet e një liste:

```
public void iterate(String[] array){
    for(int i = 0; i < array.length; i++){
        System.out.println(array[i]);
    }
}
```

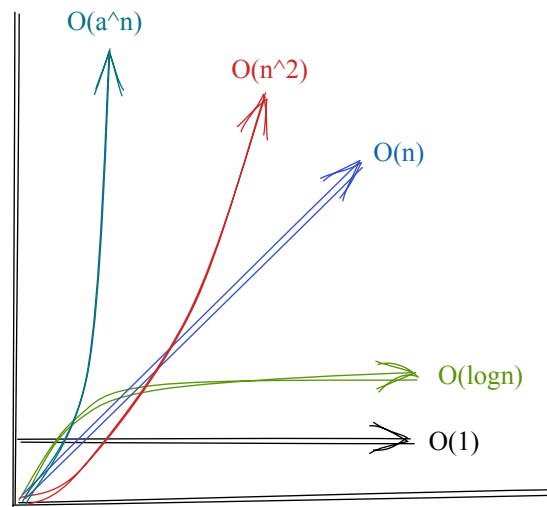
Ky algoritëm, varësisht nga gjatësia n e listës, i bën n operacione apo një funksion linear i n . Në këtë rast themi se ky algoritëm ka kompleksitet të kohës **$O(n)$** . Mirëpo nëse kemi rastin më poshtë:

```
public void iterate(String[] array){
    for(int i = 0; i < array.length; i++){
        for(int j = 0; j < array.length; j++)
            System.out.println(array[i] + array[j]);
    }
}
```

Tani, algoritmi kryen $n * n = n^2$ operacione kur ka hyrje n , andaj themi se algoritmi ka kompleksitet të kohës **$O(n^2)$** , dhe qartazi është më i ngadaltë se një algoritëm që funksionon në kohë **$O(n)$** . Nëse analizojmë algoritmin:

```
public void iterate(String[] array){
    for(int i = 0; i < array.length; i++){
        for(int j = i; j < array.length; j++)
            System.out.println(array[i] + array[j]);
    }
}
```

Shohim se kemi $n * (n + 1) / 2 = n^2/2 + n/2$ operacione. Sa është kompleksiteti tani? Qartazi, do të jetë fuqia më e madhe e polinomit, pra **$O(n^2)$** . Më poshtë shohim vizualisht krahasimet e kompleksiteteve të ndryshme:



Vërejmë se koha konstante është më e mira dhe pas saj vjen $\log n$ i njohur si kompleksitet logaritmik. Kompleksitet normal është n i cili quhet kompleksitet linear. Kompleksitet mjaftueshëm i shpejtë është $n \log n$ dhe pas tij vijnë n^2 , n^3 , etj. Kompleksiteti më i keq është ai eksponencial a^n i cili është i pafavorshëm për çfarëdo aplikimi.

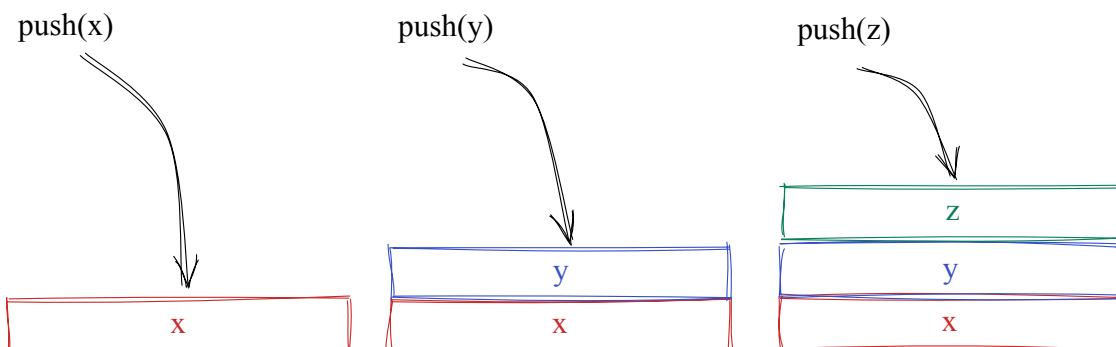
Vlen të përmendet se me të njejtën logjikë mund të masim edhe kompleksitetin e *hapsirës* sepse disa algoritme shfrytëzojnë hapsirë ekstra për të kryer punën e tyre. Shumicën e rasteve kur rritet kompleksiteti i hapsirës, zvogëlohet ai i kohës. Pra, kursejmë kohë duke shfrytëzuar më shumë memorie.

Llogaritja e kompleksitetit në shumicën e rasteve është e thjeshtë, mirëpo në algoritme rekursive apo shumë të komplikuar është problem të vihet në përfundim aq thjeshtë për kompleksitetin e tyre andaj duhet të analizohet mirë me aparat matematik.

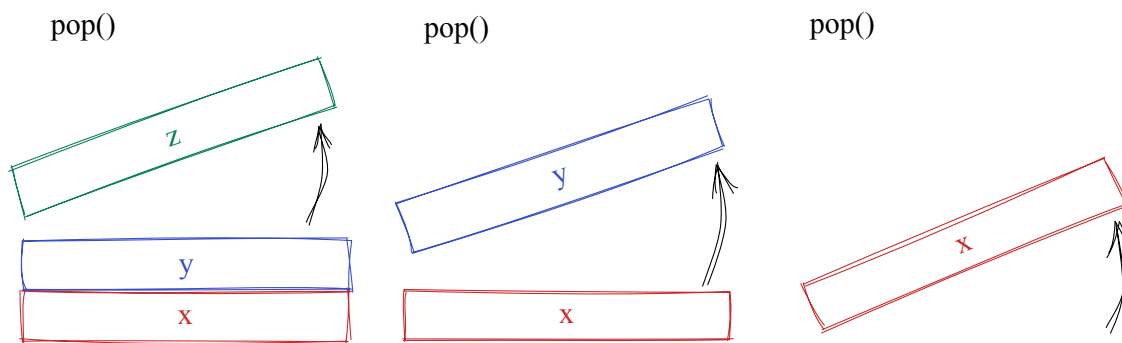
Stacks

Vargjet dhe Vargjet Dinamike ishin struktura të mira për reprezentimin e të dhënave si një listë. Mirëpo ky reprezentim i të dhënave jo gjithmonë reflekton saktë lidhjen mes të dhënave. Disa të dhëna nuk kanë renditje logjike, disa nuk lejojnë qasje në elemente në çfarëdo renditje, e disa kanë vetëm vlera unike. Për t'ju përshtatur këtyre kërkesave, struktura të tjera abstrakte të të dhënave ekzistojnë.

Një **Stack** është një strukturë e cila modelon elementet të cilat vihen njëra mbi tjetrën. Ideja është se elementet mund të vendosen një nga një me anë të një operacioni, si më poshtë:



Qasja në elemente tani bëhet në renditjen në të cilën janë vendosur. Nuk mund t'i qasemi `y` pa ju qasur së pari `z`. Pra qasja në elemente shkon sipas rregullës Last In First Out (LIFO), si më poshtë:



Pra, më formalisht, një Stack na ofron operacionet:

- `push(element)` për të shtuar një element në maje
- `pop()` për të larguar elementin në maje
- `peek()` për të shikuar elementin në maje pa e larguar atë

Modelimi i Stack-ut mund të bëhet në mënyra të ndryshme. Ne do ta bëjmë një implementim të thjeshtë duke përdorur një Varg Statik. Duke e ruajtur indeksin e elementit në maje, ne e dimë ku fillon dhe mbaron Stacku:

```
//për thjeshtësi, supozojmë se ky Stack ruan vetëm vlera String
public class Stack{
    //elementet e Stackut
    private String[] elements;

    //indeksi i elementit në maje, fillimisht -1
    private int peakIndex = -1;

    //krijojmë stack me madhësi të kufizuar
    public Stack(int capacity){
        this.elements = new String[capacity];
    }

    //shtojmë elementin e ri duke u siguruar se ka vend në varg
    public void push(String element) throws StackOverflowException{
        if(peakIndex == elements.length - 1){
            throw new StackOverflowException();
        }

        this.elements[++peakIndex] = element;
    }

    //kthejmë dhe largojmë elementin në maje duke u siguruar se ka të paktën një
    element në varg
    public String pop() throws StackUnderflowException{
        if(peakIndex == -1){
```

```

        throw new StackUnderflowException();
    }
    String element = this.elements[peakIndex];
    this.elements[peakIndex--] = null;
    return element;
}

//kthejmë elementin në maje duke u siguruar se ka të paktën një element në varg
public String peek(){
    if(peakIndex == -1){
        return null;
    }
    return this.elements[peakIndex];
}
}

```

Mund të mendoni se pse na duhet një strukturë kaq e kufizuar me kaq pak operacione, mirëpo nganjëherë më pak është më mirë. Me anë të Stackut mund të modelojmë funksionin *Undo* apo historinë e Browserit. Mund të modelojmë renditjen e vizatimit të dritareve në ekran e gjithashtu navigimin në aplikacione mobile. Jashta këtyre përdorimeve të qarta, mund ta shfrytzojmë edhe si pjesë e shumë algoritmeve.

Reverse Array

Me këtë shembull të thjeshtë mund të kuptojmë funksionimin e Stackut edhe pse nuk është zgjidhje efikase. Ideja është të vendosim elementet e array një nga një në Stack dhe pastaj ti largojmë një nga një duke i vendosur në pozicionet e tyre:

```

String[] array = new String[5];

Stack<String> stack = new Stack<>();

for(int i = 0; i < array.length; i++){
    stack.push(array[i]);
}

for(int i = 0; i < array.length; i++){
    array[i] = stack.pop();
}

```

Parenthesis Checking

Ky problem është interesante dhe paraqitet shpesh në parserë të gjuhëve programuese. Si e din Java që një `{` mungon në rreshtin 122, apo se ka problem me kllapa? Ky quhet problemi i kontrollimit të kllapave dhe zgjidhet shumë thjeshtë me anë të Stackut. Shikojmë një seri kllapash:

1. `{() }`
2. `{[]}`
3. `[()]`

Cila nga këto nuk është valide? Nëse shikojmë nga afër vërejmë se 2 nuk është valide. Validitetin mund ta kontrollojmë me një Stack, ku vendosim kllapat hapëse `{`, `[`, `(` në stack, dhe sa herë që hasim një kllapë

mbyllëse kontrollojmë se a është kllapa e fundit e hapur e të njejtit tip. Nëse mbërrijmë në fund të stringut dhe Stacku mbetet i zbrazur, stringu është valid.

```
public boolean validParentheses(String parentheses){
    Stack<Character> stack = new Stack<>();

    for(int i = 0; i < parentheses.length; i++){
        char character = parentheses.charAt(i);

        if(isOpening(character)){
            stack.push(character);
        }
        else if(isClosing(character)){
            if(stack.empty())
                return false;

            if(!areMatching(stack.pop(), character))
                return false;
        }
        else
            return false;
    }

    return stack.empty();
}

private boolean isOpening(char parenthesis){
    return parenthesis == '{' || parenthesis == '[' || parenthesis == '(';
}

private boolean isClosing(char parenthesis){
    return parenthesis == '}' || parenthesis == ']' || parenthesis == ')';
}

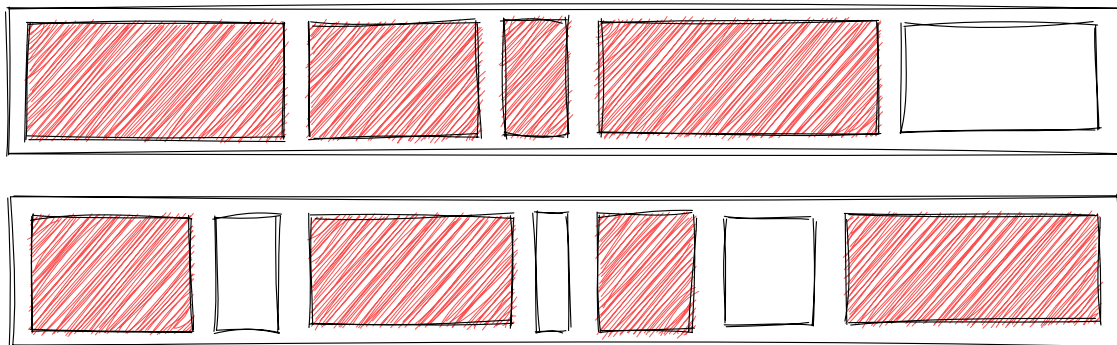
private boolean areMatching(char opening, char closing){
    return (opening == '{' && closing == '}') ||
           (opening == '[' && closing == ']') ||
           (opening == '(' && closing == ')');
}
```

Listat e Lidhura

Vazhdojmë të vërejmë se me anën e strukturave të të dhënave mund të krijojmë algoritme eficiente për zgjidhjen e shumë problemeve. Në këtë seksion do të diskutojmë për **Linked Lists** (*Listat e lidhura*) të cilat ofrojnë zgjidhje interesante për një mori problemesh.

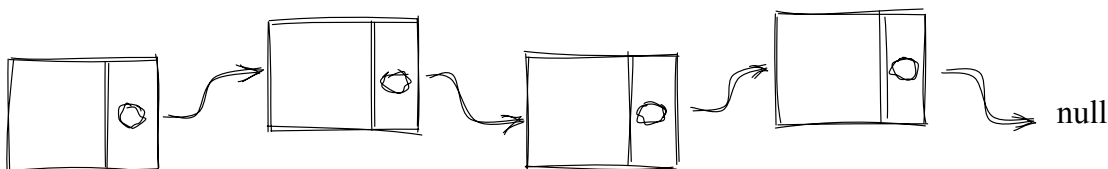
Strukturat e të dhënave si Vargjet Statike, e gjithashtu Vargjet Dinamike dhe Stacks të cilat modelohen përmes vargjeve statike, të gjitha e kanë një problem: *Ju nevojitet një bllok i pandërprerë i memories për të ruajtur të dhënat*. Kur një varg përmban 10000 elemente të tipit `int`, i duhen `40000byte ~ 40kB` memorie e pandërprerë. Në ambiente me memorie të vogël, apo shumë të ngarkuar, në memorie paraqitet problemi i **fragmentimit**. Logjikisht ky term kuptohet se memoria është e nxënë në vende të shpërndara, dhe është problem të gjenden numër i madh i qelizave të lira *fqinje*. Pra, na shfaqet problemi se edhe pse ka

mjaftueshëm hapësirë për të ruajtur të dhënat, hapësira nuk është në një bllok të vetëm - andaj nuk mund ta shfrytëzojmë.



Në foto vërejmë dallimin mes një memorie të shëndetshme dhe asaj të fragmentuar. Edhe pse madhësia e memories së lirë është e njëjtë, ajo është e shpërndarë nëpër të.

Listat e Lidhura janë një strukturë e të dhënave në të cilën secili element përveq të dhënave të veta përmban adresën e elementit tjetër në radhë:



Në këtë formë mund të iterojmë nga elementi i parë deri tek i fundit. Elementi i parë zakonisht quhet **head** dhe është pika ku fillon gjithçka. Vërejmë se për të shenjëzuar elementin e fundit, elementi i fundit drejton tek një vlerë `null` dhe shenjëzon se nuk ka elemente tjera. Listat e lidhura kanë një të mirë se elementet mund të shtohen dhe të fshihen edhe nga mesi i listës pa afektuar elementet e tjera - pra s'ka nevojë të kopjohen në një vend tjetër. Nëse dihet vendi se ku do të shtohet, këto operacione marrin kohë konstante. T'i qasemi një elementi, nga ana tjetër, nuk është aq e thjeshtë. Nëse dihet indeksi i elementit ne duhet të iterojmë nga koka e deri tek ai element për të marrë atë. Për këtë arsye qasja është **$O(n)$** . Shohim se si mund të modelohet konceptualisht një Listë e Lidhur.

Elementet e një Liste të Lidhur zakonisht quhen *nodes*

```
class LinkedList{
    private Node head;

    //metoda që shton një node të ri
    public void set(String data){

    }

    //metoda që kthen të dhënat në indeksin e caktuar (pa kontrolluar për vlera
    ekstreme)
    public String get(int index){
        Node temp = head;
```

```

    for(int i = 0; i < index; i++){
        temp = temp.getNext();
    }

    return temp;
}

//metoda që ndryshon të dhënat në indeksin e caktuar
public String update(int index, String data){

}

//metoda që ndryshon të dhënat në indeksin e caktuar
public String delete(int index){

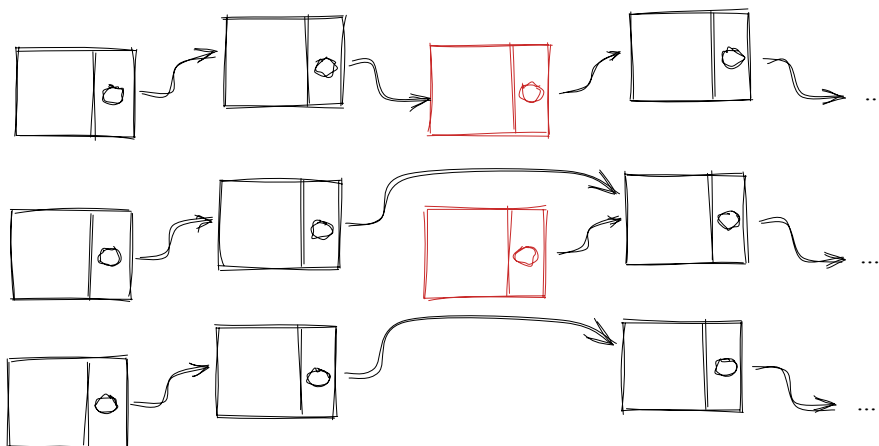
}

//për thjeshtësi, supozojmë se të dhënat janë të tipit String
private class Node{
    private String data;
    private Node next;

    //getters, setters
}
}

```

Vërejmë se implementimi e mban të fshehur klasën `Node` pasi përdoruesit e jashtëm nuk interesohen për implementimin e brendshëm. Si të shtojmë një element të ri në listë? Konceptualisht duhet të marrim elementin e fundit dhe të vendosim atributin `next` të tij që drejton tek elementi i ri. Koncept interesante është fshirja e një elementi në mes të listës. Mjafton të marrim referencën e elementit para ta tij dhe të vendosim atributin `next` duke e injoruar pasardhësin, si më poshtë:



```

//duke injoruar kontrollimin e indeksit për vlera ekstreme
public String delete(int index){
    Node before = this.get(index - 1);

```

```
before.setNext (before.getNext () .getNext () );
}
```

Listat e lidhura kanë shumë variacione. Le të shohim disa prej tyre:

- **Singly Linked List:** Kjo është lista sic e diskutuam deri tani, ku secili element ka referencën e elementit tjetër në rradhë.
- **Doubly Linked List:** Një lloj i listës së lidhur ku secili element ka referencë për tek elementi paraardhës dhe elementi pasardhës. Në këtë formë mund të iterojmë edhe tek para edhe tek mbrapa. Normalisht kjo e rrit edhe madhësinë që zihet në memorie.
- **Circular Linked List:** Një lloj i listës së lidhur ku elementi i fundit nuk ka pasardhës vlerën `null` por elementin e parë, në këtë formë duke krijuar një cikël.
- **XOR Linked List:** Një Singly Linked List e cila na lejon iterimin edhe tek para edhe tek mbrapa, duke gjetur adresat përmes operacioneve me bitë.

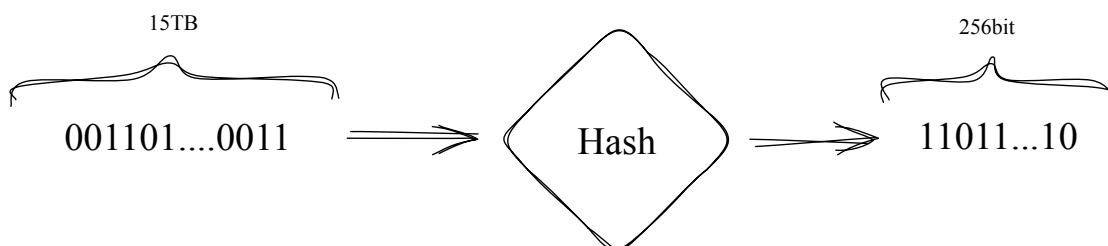
Listat e lidhura kanë përdorime të shumta, përdoren nga sistemet operative për të menagjuar proceset dhe threads, apo për të menagjuar state. Gjithashtu listat e lidhura përdoren për të modeluar struktura të tjera si Listat Dinamike dhe Grafet.

Hash-Funksionet dhe Hash-Tabelat

Paramendoni se keni një listë të parenditur me emra të personave. Si do ta dinit se a ekziston emri *Beni* në atë listë? Mënyra e parë do të ishte të shikonim të gjithë emrat derisa të gjejmë emrin ose të përfundojë lista. Kjo qasje na jep kompleksitet $O(n)$. Nëse kemi për të përsëritur këtë operacion shumë herë, mund të i'a vlejë të *rendisim* listën me një algoritëm të shpejtë me kompleksitet $O(n \cdot \log n)$, dhe pastaj të bëjmë kërkim me *Binary Search* në kohë $O(\log n)$. Kjo me të vërtetë e rrit shpejtësinë. Mirëpo a mund ta bëjmë edhe më shpejtë? Cuditërisht, me anën e **Hash Table** mund ta realizojmë këtë.

Hash Funksionet

Një Hash Funksion është një funksion i cili si hyrje e merr një string bitësh dhe si dalje e ka një string të bitëve me *gjatësi të fiksuar*.



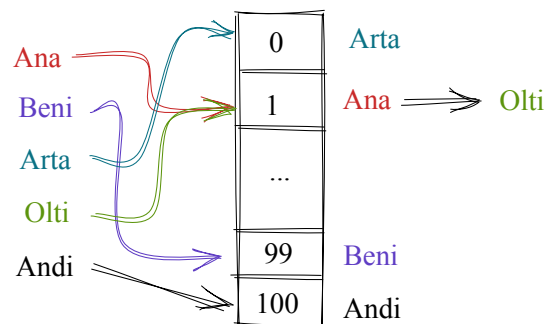
Një hash funksion disa veti kryesore ku më të rëndësishme janë se:

- Një ndryshim i vogël në hyrje shkakton një ndryshim të madh në dalje
- Hash i dy vlerave të njejta është i njejtë
- Dy hyrje të ndryshme shumë vështirë japin të njejtën dalje - ky rast quhet kolizion

Hash funksione të njohura janë *MD1* apo *SHA256* i cili shfytëzohet shumë në teknologjinë *Blockchain*.

Hash Tabelat

Hash Tabela është strukturë e të dhënave ku secili element përmban një dyshe `(key, value)`. Secili element e ka një celës unik dhe atij i qasemi përmes celësit. Hash Tabela i ruan elementet krejtësisht varësisht prej celësit të tyre. Ajo i llogarit hash funksionet e secilit celës dhe i konverton ato në *indekse të një vargu*. Kur ndodh një kolizion ka mënyra të ndryshme për grupimin e elementeve nëpër *buckets*. Përfundimisht, kur kontrollojmë se a ekziston emri *Beni* në listë, llogaritet `hash(Beni)` dhe dihet se në cilin *bucket* gjendet elementi, pastaj kërkohet në mënyrë lineare në atë bucket se a është emri aty. Kjo merr **kohë konstante** pasi numri i buckets dhe gjatësia e tyre nonstop kontrollohen dhe rivendosen varësisht prej *load factor*.



Hash Tabela përdoret për të modeluar vargjet asociative, bashkësitë, për të implementuar cache, indeksimi i databazës, etj. Do të shohim se si të përdorim në nivelin konceptual Hash Tabelën në Java.

Cache i thjeshtë në Java

Supozojmë se kemi një klasë e cila i merr të dhënat nga një link i caktuar, sikur browseri. Supozojmë se këto të dhëna ruhen në tipin String dhe se kanë për t'u analizuar më vonë nga sistemi jonë. Një zgjidhje do ishte kjo:

```
public class WebClient{
    private RestTemplate client = new RestTemplate();

    public String getContent(String url){
        return client.getForEntity(url, String.class);
    }
}
```

Kjo klasë mjafton për të marrë të dhëna nga një URL. Mirëpo cka nëse na duhet të shikojmë të njejtin URL shpesh dhe na duhet në vende të ndryshme të programit. A ja vlen që të mundojmë rrjetin kur thjesht mund t'a ruajmë rezultatin për një kohë të caktuar. Këtë koncept e quajmë *caching* dhe ja si do e bënim në këtë rast:

```
public class WebClient{
    private RestTemplate client = new RestTemplate();
    private HashMap<String, CachedData> cache = new HashMap<>();
    private long cacheExpiresAfterMillis = 1000 * 60;

    public String getContent(String url){
```

```

    CachedData cachedData = cache.get(url);
    long currentTime = System.currentTimeMillis();

    if(cachedData == null || cachedData.hasExpired(currentTime)){
        String data = client.getForEntity(url, String.class);
        cache.put(url, new ChachedData(data, currentTime +
cacheExpiresAfterMillis));
        return data;
    }
    else{
        return cachedData.getData();
    }
}

private class CachedData{
    private String data;
    private long expiresAt;

    public CachedData(String data, long expiresAt){
        this.data = data;
        this.expiresAt = expiresAt;
    }

    public boolean hasExpired(long time){
        return expiresAt < time;
    }

    //getters, setters
}
}

```

Në këtë formë klienti ynë mund t'i mbajë të dhënat në cache për kohë të specifikuar dhe t'a rrisë performancën e aplikacionit. Hash Tabelat janë strukturë e famshme e të dhënave dhe kanë përdorim shumë të gjerë andaj ja vlen të studiohen edhe më thellë.

Për lexim të mëtutjeshëm

Ky material ishte vetëm një njoftim i shkurtër me strukturat e të dhënave dhe algoritmet, dhe disa implementime bazike në kod. Një studim i thellë i tyre nuk ishte qëllim fillestar i këtij materiali. Pavarësisht kësaj, ne ju sugjerojmë shumë që të vazhdoni studimin e tyre dhe të gjeni aplikime të vlefshme të tyre gjatë karrierës suaj. Më poshtë do të listojmë disa prej burimeve ku mund të mësoni më shumë dhe t'i aplikoni njohuritë tuaja.

Mësimi teorik rigoroz zakonisht bëhet prej librave, andaj ju sugjerojmë librat më poshtë:

- *Grokking Algorithms: An Illustrated Guide for Programmers and Other Curious People*
- *Algorithms in a Nutshell: A Practical Guide*
- *Introduction to Algorithms, 3rd Edition*

Jashta librave, aplikimi i njohurive tuaja është hapi tjetër për të kuptuar konceptet dhe aplikimet e tyre.

Resurset më poshtë përmbajnë me qindra mijëra probleme interesante për tu zgjidhur me anën e strukturave të të dhënave dhe algoritmeve, e të cilat kanë shumë përkrahje nga komuniteti:

- *LeetCode*
- *HackerRank*

Nëse preferoni të mësoni prej ligjëratave dhe videove online, mund të vijoni kurse pafund prej *Udemy* apo *Coursera*.

This document has been developed with the financial assistance of Austrian Development Cooperation and implemented by WUS Austria (Lead Consortia). The views and opinions expressed herein, can in no way reflect the official opinion of the Austrian Development Cooperation nor the opinion of the implementing partner.
