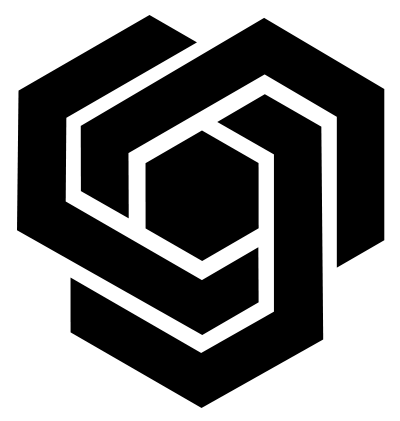
**Технически университет-София**



**Дисциплина:**

**„Синтез и анализ на алгоритми - САА”**

Изработил: Христо Ненков Проверил:

Фак.№: 121223001

Факултет: ФКСТ

Група: 45

Дата: 25.11.2024 /……………………………/

1. Въведение

По дадените задачи 14 и 44, съответно за имплементиране на двойно свързан списък чрез езика за програмиране – C и визуализация на алгоритъм на Хуфман на същия език, е използвана средата JetBrains – CLion.

Задача 14 е обяснена в дълбочина и има за цел да обясни всеки един ред от кода по начин, достъпен за човек незапознат с езика C и неговите възможности и синтаксис.

Задача 44, от своя страна, поради по – голямата си сложност, очаква читателя да разполага с базови знания в езика C. В тази задача се обяснява много повече логиката на алгоритъма, отколкото самият синтаксис.

В имплемантацията на задача 14, използваме частния случай – двойно свързан списък от целчислени числа с индексация, започваща от 0.

И към двете задачи са представени презентации, изпозвани от мен, за представяне на курсовата работа, както и целият код от програмата, заедно със обясненията към него.

**Линк към двете задачи има тук:**

https://github.com/hristo-nenkov-6/Algorithms-and-data-structues

1. Задача 14 – Двойно свързан списък
2. Въведение

Дройно свързаният списък или Doubly Linked List е структура от данни съставени от отделни елементи (Nodes), всяка заемаща неопределена клетка памет. Всеки Node съдържа данни, както и информация за адреса на предишния и следващия Node.

1. Дефиниране на структурата

Двойно свързаният списък е съставен от Nodes, които са свързани помежду си и съдържат определени данни в себе си. В структурата имаме дефинирани съответно:

* Указател към следващият Node- next
* Указател към предишният Node - previous
* Информация – data

Използваме и typedef, за да можем да създадем тип Node, който по – късно използваме при заделяне на памет и при изпозването в main функцията.

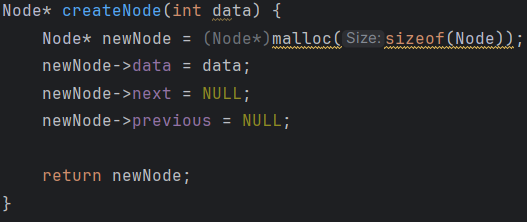
За да бъде по – ясно, защо **head** елемента, който се подава във всеки метод е указател или двоен указател, е нужно да си представим паметта. Един масив може да бъде зададен по начините **arr[]** и **\*arr**, като и в двата случая достъпването на **arr**, сочи към първия елемент. Масива, обаче, е структура, която представлява последователни клетки от памет, на която ние знаем първия елемент и чрез прости операции можем да достигнем до другите.

Двойно свързания списък е структура, която представлява свързани Node структури, чрез указатели. Паметта не е последователна, а във main метода ние подаваме на манупуиращите методи, указател към клетка памет. Както знаем, параметрите на манипулиращите методи са просто копия на данните подадени при извикване. Тъй като ние искаме нашите методи да променят реланите клетки от паметта, използваме двойни указатели като аргументи.

1. Методи за добавяне на елементи към свързания списък
2. **Метод за създаване на нов Node**

При създаване на нов Node, използваме вградената в библютеката <stdlib.h>, функция malloc, на която съответно подаваме параметър – памет, заемана от една променлива от тип Node.

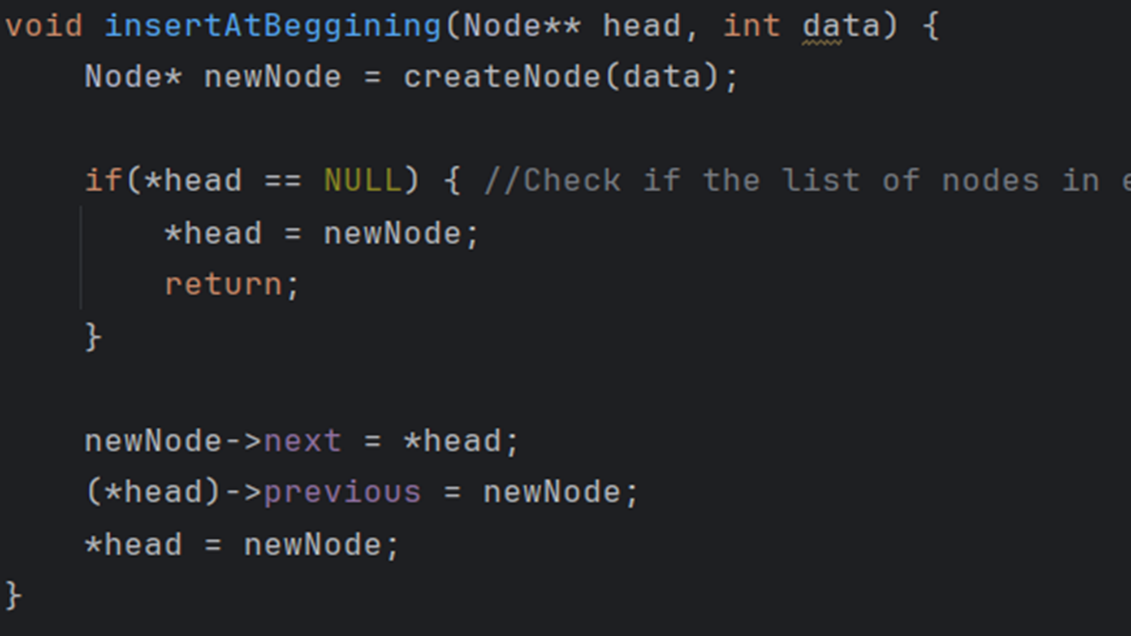
Съответно **Node\* createNode(int data)**, приема стойността, която искаме да присвоим на дадения Node и задава на атрибутите next и previous, стойност NULL. Този метод връща новосъздадения Node.



1. **Метод за добавяне на елемент на позиция 0**

Първоначално създаваме нов Node - newNode със вече дефинираната ни функция **createNode(int data)**.

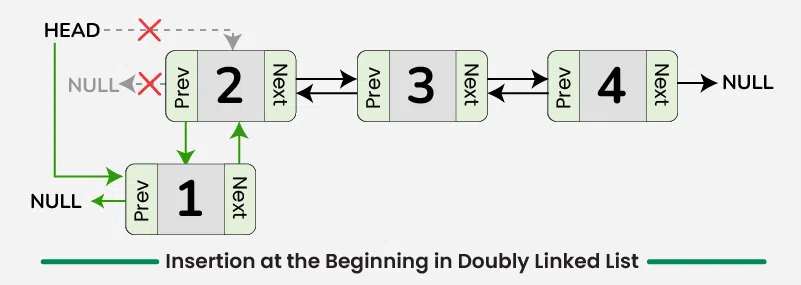
Методът **void insertAtBeggining(Node\*\* head, int data)** приема двоен указател към Node - head и data. Съответно head е първият елемент от двойно свтрзаният списък. Тъй като обаче нашата структура се състои от непоследователни клетки от паметта, ние работим само с указатели. По – късно ще видим в main метода, че head елемента там се задава с указател, следователно във всеки един допълнителен метод, който иска да манипулира нашият двойно свързан списък, ние трябва да подаваме двоен указател като аргумент, за да можем да правим промени към реалната ни струкура в паметта.



**head** елемента има указател **previous = NULL** и **next** равно на друг Node или на **NULL**

В тялото на метода, проверяваме дали той е празен (като проверяваме дали подаденият ни първи елемент е NULL) и след това правим следните операции:

* Модифицираме указателя **next** на **newNode** да сочи към досегашният първи елемент – **head**
* Модифицираме указателя **previous** на **head** да сочи към новия първи елемент – **newNode**
* Задаваме нов **head** елемент в лицето на newNode



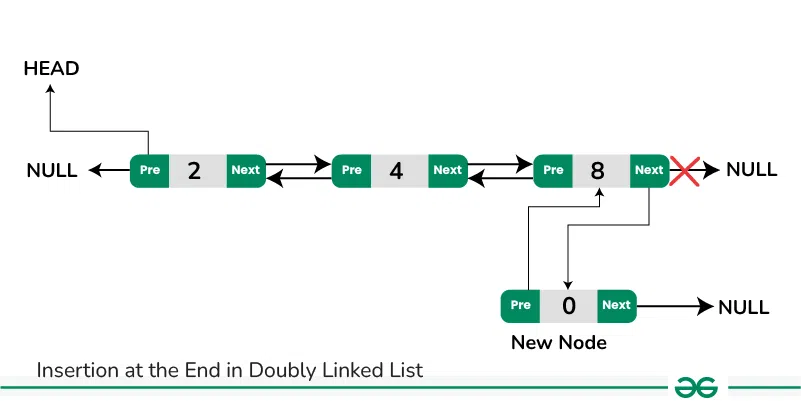
1. **Метод за добавяне на елемент в края**

Методът **void insertAtEnd(Node\*\* head, int data)** също създава нов Node - newNode и проверява за празен двойно свързан списък.



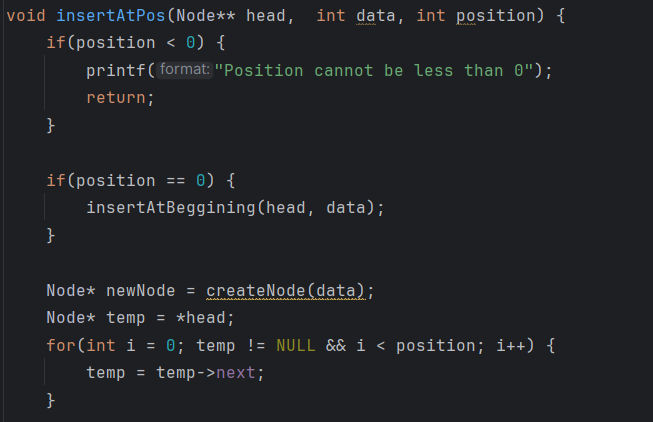
Той обаче прави следните операции:

* Стигаме до последния елемент със while цикъл, проверяващ дали следващият Node не е **NULL** (съответно дали не сме стигнали последния Node). Това нещо става с помощта на нов указател temp от тип Node, който сочи към head. По този начин променяме temp и елемента, към когото сочи без да променяме нашият двойно свързан списък
* Модифицираме **next** на temp да сочи към newNode(новият ни последен елемент)
* Модифицираме **previous** наnewNode да сочи към temp(вече предпоследен елемент)



1. **Метод за добавяне на елемент на дадена позиция**

Методът **insertAtPos(Node\*\* head, int data, int position)** приема още един допълнителен параметър position, който представя позицията, на която искаме да добавим нов елемент.

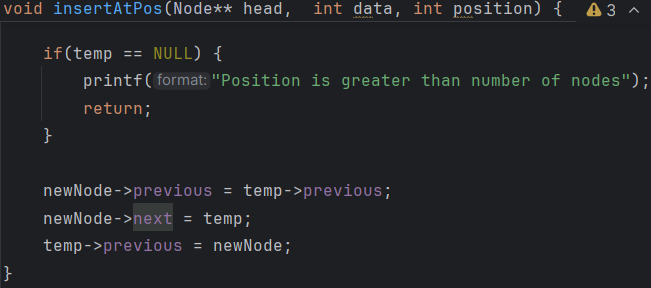


**Другата част на метода е в следващата снимка**

В тялото не метода проверяваме дали подадената позиция е валидна (по – голяма или равна на 0). При случай, че подадената позиция е 0, извикваме **insertAtBeggining** метода.

След това създаваме нов Node и указател temp и извършваме операция, подобна на търсенето на крайния елемент в **insertAtEnd** функцията. Този път обаче използваме for цикъл и проверяваме дали не сме достигнали несъществуващия елемент, който е равен на NULL, към когото сочи последия елемент. В същото време следим и дали не сме достигнали до търсената от нас позиция.

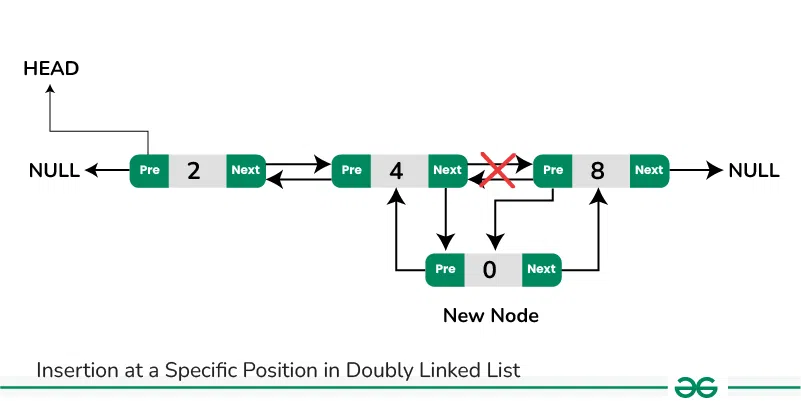
В добрия случай temp е елемента на позицията, която желаем.



След това проверяваме дали **temp = NULL** (дали сме излезли от границите на нашия списък) и съответно ако не сме, правим следните операции:

* Модифицираме **previous** на **newNode** да сочи към същия елемент, към който сочи и **previous** на **temp**
* Модифицираме **next** на **newNode** да сочи към **temp**
* Модифицираме **previous** на **temp** да сочи към **newNode**

Рисунката в тази ситуация би била много полезна, поради по – сложните операции.



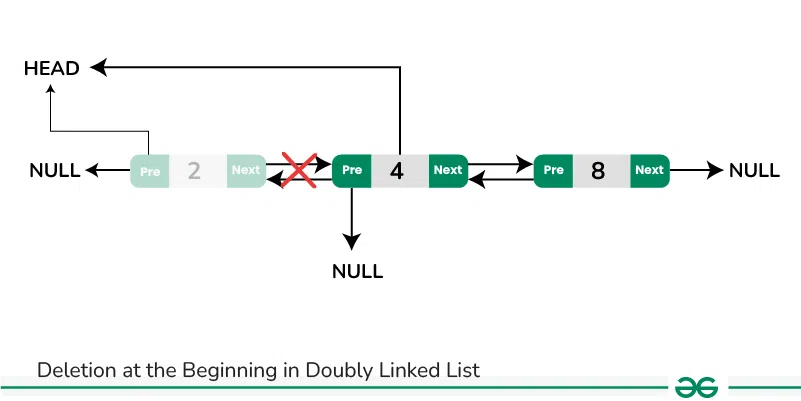
1. Методи за изтриване на елементи от свързания списък
2. **Метод за изтриване на първия елемент**

Методите за изтриване на елементи не приемат data, а само началния елемент и съответно позиция в дадения случай. Проверяваме дали списъкът ни не е празен. Този път подхождаме по различен начин, отново използвайки temp.

****

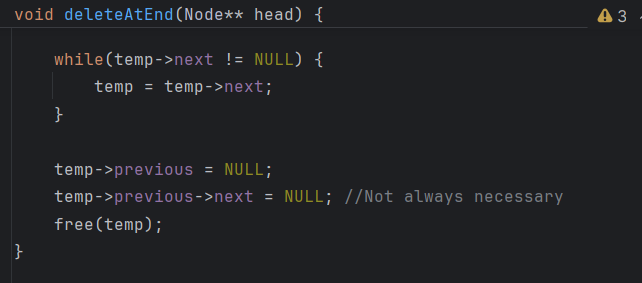
Тък използваме следните операции:

* Модифицираме temp да сочи към първия елемент.
* Тъй като премахваме първия елемент, преместваме head (подадения, първи елемент на списъка) да сочи към втория елемент.
* Проверяваме дали head(вече съответно втория елемент) е NULL – тази проверка е за да проверима дали не се съдържа само един елемент във списъка.
* Ако има повече от 1 елемент, модифицираме previous на втория елемент да сочи към NULL – по този начин го правим първи.
* Освобождаваме паметта, в която се намира първия елемент (към когото сочи temp)

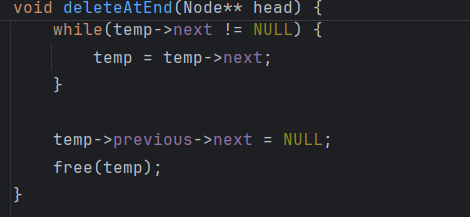


1. **Метод за изтриване на последния елемент**

Отново проверяваме дали листа ни е празен (първият елемент head = NULL), и съответно ако не е, правим втора проверка. Тази проверка, с помощта на допълнителния указател от тип Node – temp, е за да бъдем сигурни, че нашия лист не съдържа само един елемент(операциите са подобни на предишния метод). Ако единствения елемент е head, следователно го премахваме и лист остава празен.



Използваме while цикъл, за да достигнем до последния елемент на свързания ни списък и след това правим следните операции:

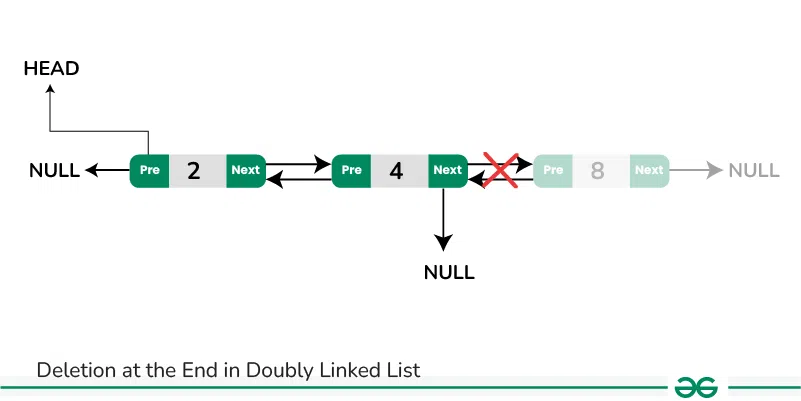


* Модифицираме previous на **temp**(последния елемент), да сочи към **NULL** – по този начин прекъсваме връзката му със свързания списък и го елиминираме.
* **temp->previous->next** е една по – сложна операция, която можем да напишем и по този начин.

**Node\* beforeLast = temp->previous;**

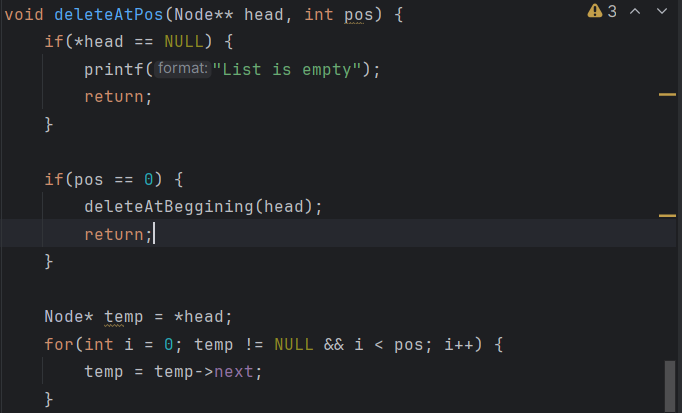
**beforeLast->next == Null;**

* Освобождаваме паметта, заемана от елемента към който сочи temp

****

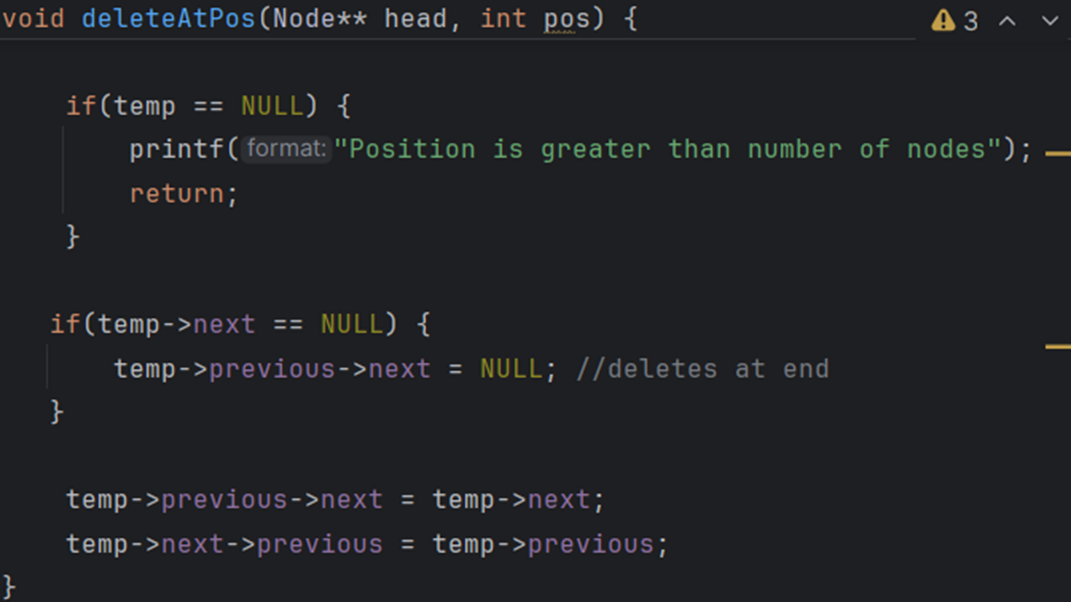
1. **Метод за изтриване на елемент на дадена позиция**

Достигаме до последния и най – сложен метод от нашия клас. Операциите от първата снимка са абсолютно идентични с тези за добавяне на елемент на дадена позиция, като единсвената разлика е, че използваме **deleteAtBeggining** вместо **insertAtBeggining**

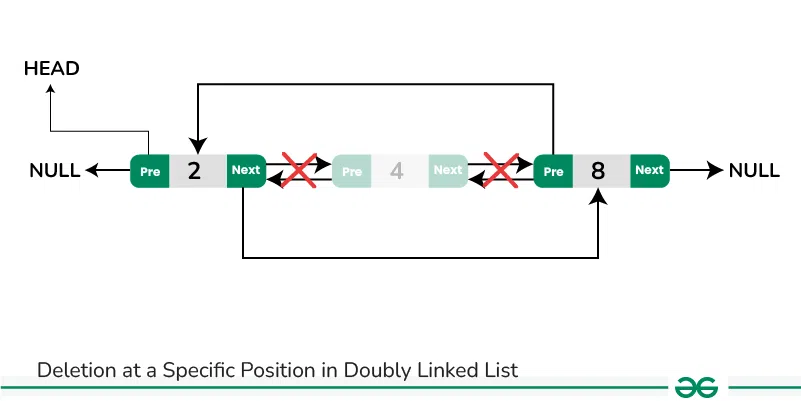


След като сме проверили дали има елементи във списъка ни, дали позицията не е 0 и вече сме достгнали до елемента на позицията, която искаме, правим отново позната ни проверка за вярност на подадената позиция.

Вече сме сигурни, че елемента ни е между 1ва и (n – 2)ра позиция на списъка, преминаваме към операциите:



* Чрез условния оператор if, проверяваме дали нашият Node не се намира на последната позиция. Ако той се намира там, имаме възмжността да извикаме вече познатия ни **deleteAtEnd** метод, но след като вече сме достигнали до последния ни елемент, просто можем да повторим операцията **temp->previous->next = NULL**(обяснено е по - нагоре)
* Операцията **temp->previous->next = temp->next**, модфицира **next** на елемента преди търсения от нас(този на позиция pos), да сочи към елемента след търсения от нас.
* Операцията **temp->next ->previous = temp-> previous**, модфицира **previous** на елемента след търсения от нас(този на позиция pos), да сочи към елемента преди търсения от нас.
* Тези две операции, свързват елемента преди **pos**, със елемента след **pos**. По този начин се премахва този Node, към който сочи temp.
* Освобождаваме паметта, заемана от елемента към който сочи temp

****

1. Методи за визуализация на двойно свързания списък
2. **Метод за последователна визуализация на двойно свързания списък**

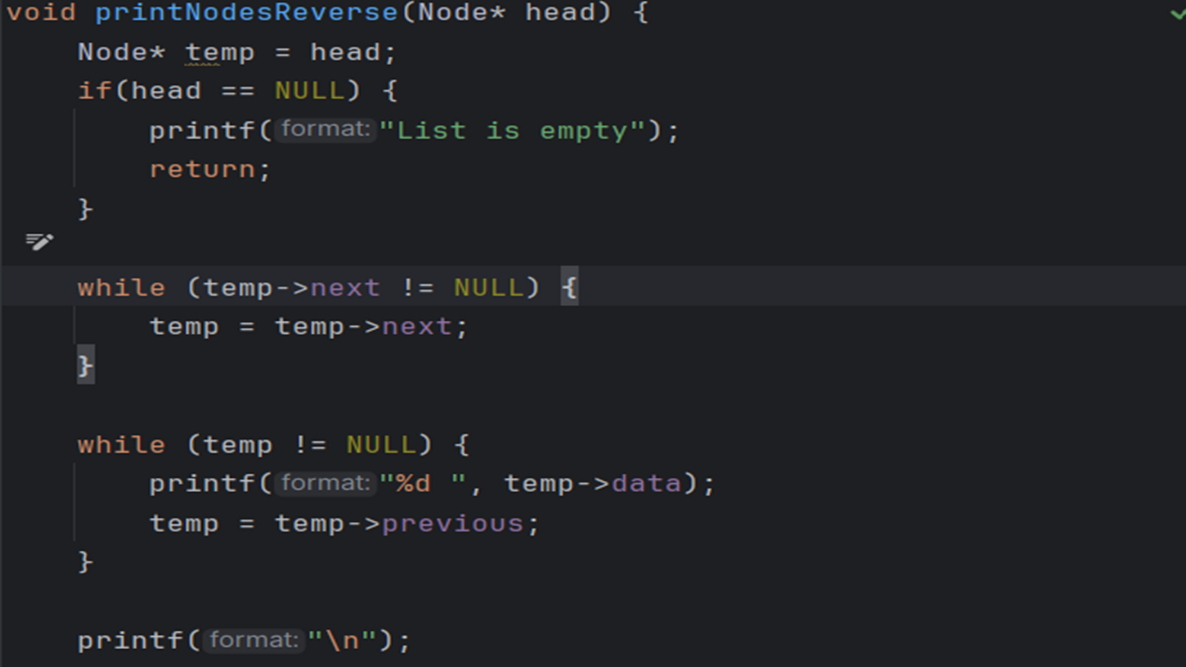
Нашият метод void **printNodes(Node\* head)**, приема единичен указател **head** от тип Node, тъй като вече няма да манупулираме нашият свързан списък, съответно се нуждаем само от копие на нашият свъран списък.



След като проврерим за празнота на масива и си създадем вече познатия указател **temp**, използваме цикъл **while**, за да принтираме информацията – **data**, съдържана във всеки един наш **Node**. Тъй като искаме да принтираме списъка последователно, използваме **next** на всеки един елемент (движим се към следващия елемент)

1. **Метод за обратна визуализация на двойно свързания списък**

Методът **void printNodesReverse(Node\* head)**, има подобна логика като на **printNodes**, като разликата е, че първоначално стигаме до последния елемент на списъка(чрез **temp**) и след това се местим по елементите наобратно – чрез използване на **previous** на **temp**

****

1. Заключение за двойно свързан списък

Двойно свързния списък е структура от данни, използвана за множество процеси както в УЕБ програмирането, така и в програмирането на ниско ниво.

В УЕБ, той би могъл да се използва за пазене на данни в КЕШ паметта на браузъра, за направление между страници и пазене на история.

В операционните системи е много често използван за записване на процесите, изпълнявани на процесора, тъй като винаги могат да бъдат погледнати предишните процеси без разбркване на данните (както би се случило при стек и опашка).

Въпреки многото си предимства, двойно свързания списък е много бавен при изтриване и добавяне на елементи в края на списъка.

1. Задача 44 – Код на Хуфман
2. Въведение

Кодът на Хуфман е алгоритъм за криптиране на символни низове в последователност от бинарни числа. Той използва Binary Search Tree за криптиране на всеки символ, което го прави и изключително бърз. Логиката, стояща зад алгоритъма, се базира на честотата на срещане на всеки един елемент в низа.

За изпълнението на алгоритъма ние се нуждаем от 2 сортитани възходящо масива, единият представящ всеки символ от низа, а другия честота му на срещане – те ще бъдат нашо импровизирано Dictionary.

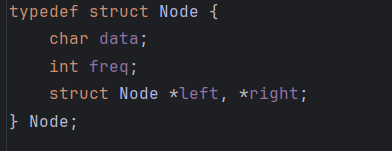
1. Дефиниране на нужните ни структури

Кодът на Хуфман се нуждае от 2 структури – Node и Tree.

1. **Структура Node – всеки елемент**

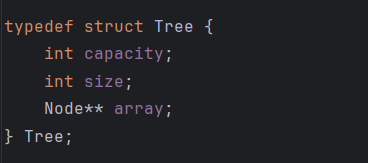
Node, съдържа в себе си:

* data – Съдържа символът, който се крие зад този Node
* freq – Съдържа честотата, на срещане а символа data
* \*left – Указател към левият Node в дървото
* \*right – Указател към десния Node в дървото

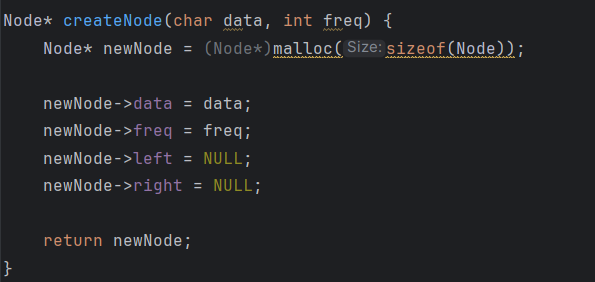


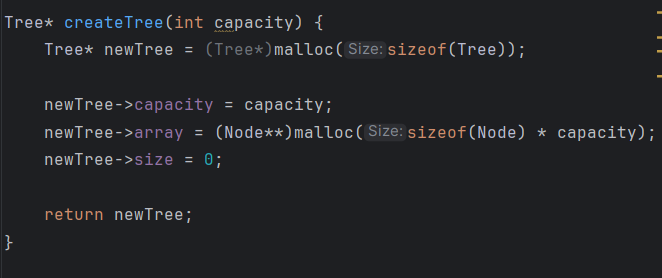
1. **Структура Tree – цялото бинарно дърво**

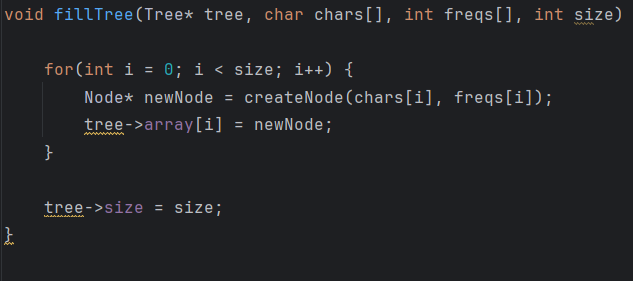
* size – Съдържа текущата реална големина на масива array
* capacity – Съдържа максималната големина, до която може да достигна дървото
* \*\*array – Масив, съдържащ указатли към всички елементи, съдържащи се в дървото. Особеното за него е, че той трябва да бъде сортиран по честотата на съдържащите се елементи преди всяка нова операция



1. Методи, създаващи структурите







1. Пример, показващ как алгоритъма работи

Сега ще разгледаме примерен низ, който ще ни покаже логиката и последователните стъпки при изпъление на този алгоритъм. Нека вземем символите и техниче честоти в примерен низ:

character Frequency

a 5

b 9

c 12

d 13

e 16

f 45

***Ще наричам комбинацията от тези***

***два масива – речник (Dictionary) от Nodes***

***Стъпка 1 – Създаване на сортиран масив от крайни елементи***

Всеки един елемент от нашия речник представлява един Node с:

* data = character[i]
* freq = Frequency[i]

Идеята на алгоритъма е да създаде Binary Tree, чиито последни деца (елементи в дървото, които нямат наследници, наричани **крайни елементи**) е всеки един Node от нашия речник.

Нашит масив трябва да бъде винаги сортиран низходящо, за да можем лесно да добяваме и премахваме елементи в края:

character Frequency

a 5

b 9

c 12

d 13

e 16

f 45

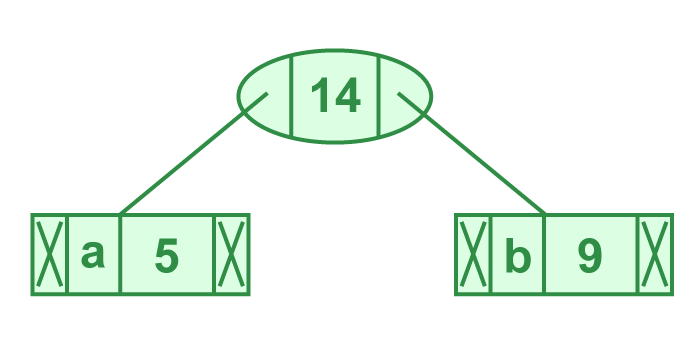
***Стъпка 2 – Взимане на двата най – малките елементи от масива***

Ще бъде обяснено по – късно с метода getSmallest

***Стъпка 3 – Комбиниране на тези два най – малки елемента***

За целта ние ще комбинираме двата елемента с най – малка честота на срещане(това са последните два елеемнта на масива) в един, който ще бъде техен родител. Този комбинран елемент няма да съдържа data, а единсвено честота на срещането – сумата на честотите на срещане на неговите два наследника. Той ни е нужен единствено за показване на пътя към краен елемент в дървото.

Тъй като ние се стремим към създаване на Бинарно дърво, в комбинирания елемент left ще бъде елемента с най – малка честота, а right – вторият с най – малка честота. Вижте картинка:



След като вече имаме комбинирания Node, ние го добавяме отново към нашия масив от елементи на последното място.

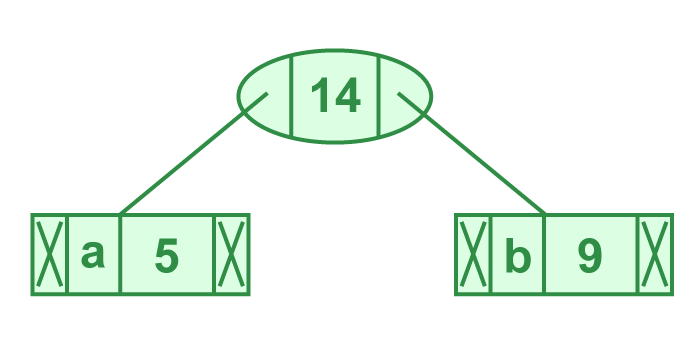
Сортираме масива в низходящ ред.

**!NB! – При премахване на последните елементи и добавянето на комбинарания, винаги трябва да намаляме (при премахване) и увеличаваме (при добавяне), size на нашия масив. Тъй като работим с масив, на който не променяме реалната дължина в паметта, array[size – 1] сочи към последния елемент на масива ни.**

***Повторение на стъпка 2 и стъпка 3 до оставане на само един елемент в масива***

Сега ще поставя последователност от снимки на това как изглежда нашият масив и съответно, как се развива нашето Бинарно дърво през стъпките

1. **Цикъл 1**



Character Frequency

f 45

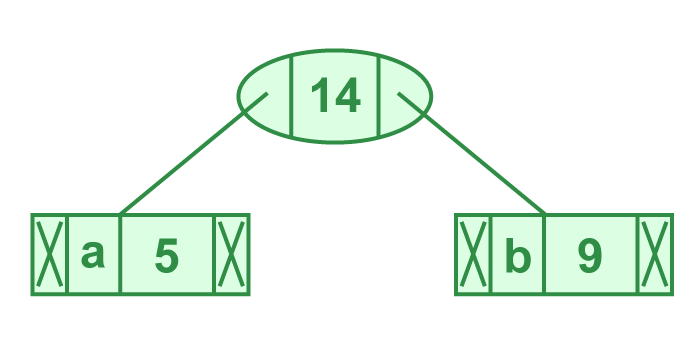
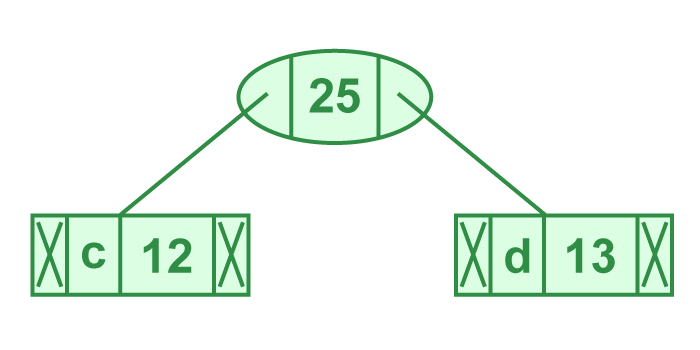
e 16

combined 14

d 13

c 12

1. **Цикъл 2**

Character Frequency

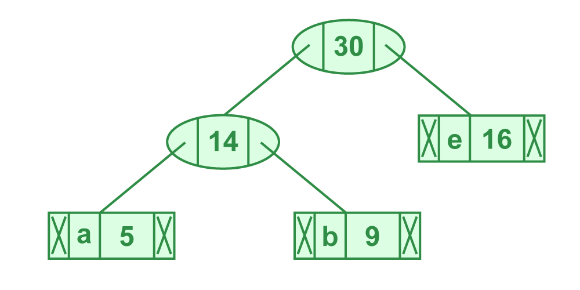
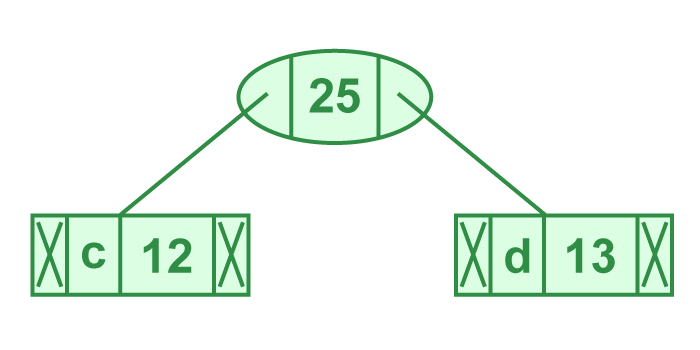
f 45

combined 25

e 16

combined 14

1. **Цикъл 3**

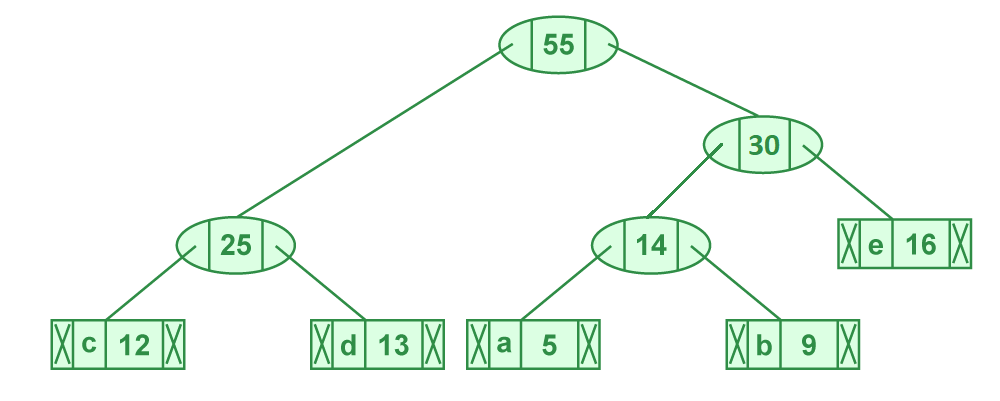
Character Frequency

f 45

combined 25

combined 30

1. **Цикъл 4**

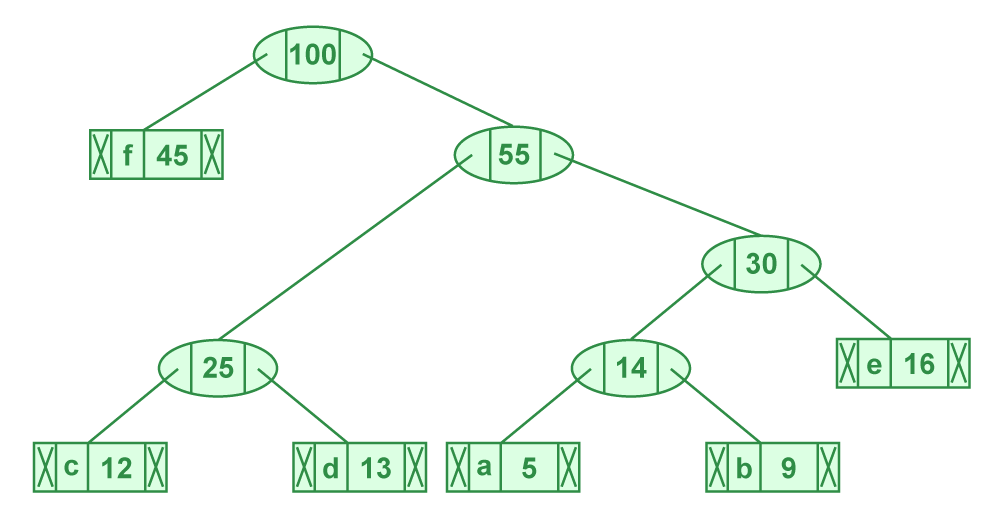


Character Frequency

f 45

combined 55

1. **Цикъл 5**

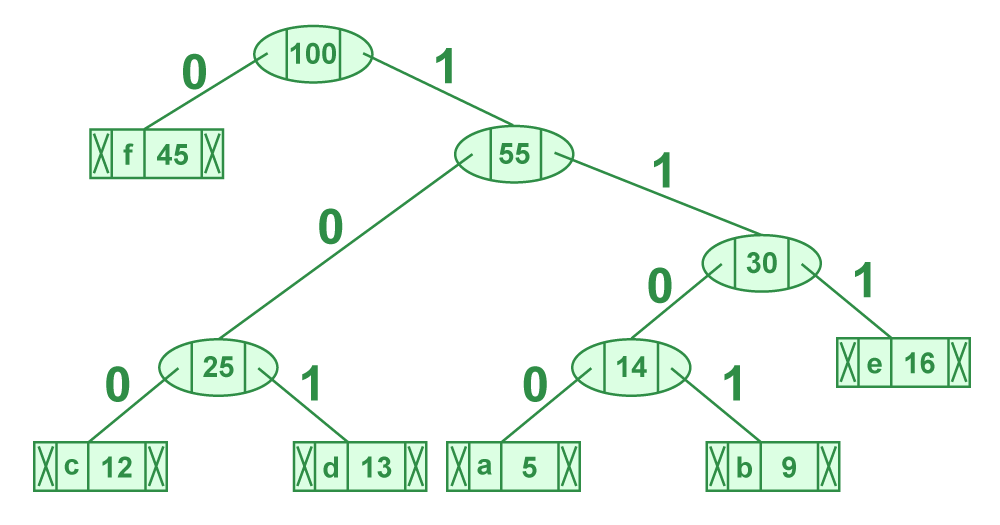


Character Frequency

combined 100

***Стъпка 4 – Принтиране на всеки един криптиран елемент***

Това е снимка на начина по който ще криптираме бинарно нашите символи:



Тръгваме от корена – елемента, който няма родител

(От горе, надолу)

* При преместване на ляво – **Пишем 0**
* При преместване на дясно – **Пишем 1**

След като сме крипирали всички елементи, можем да видим и как изглеждат те:

character code-word

f 0

c 100

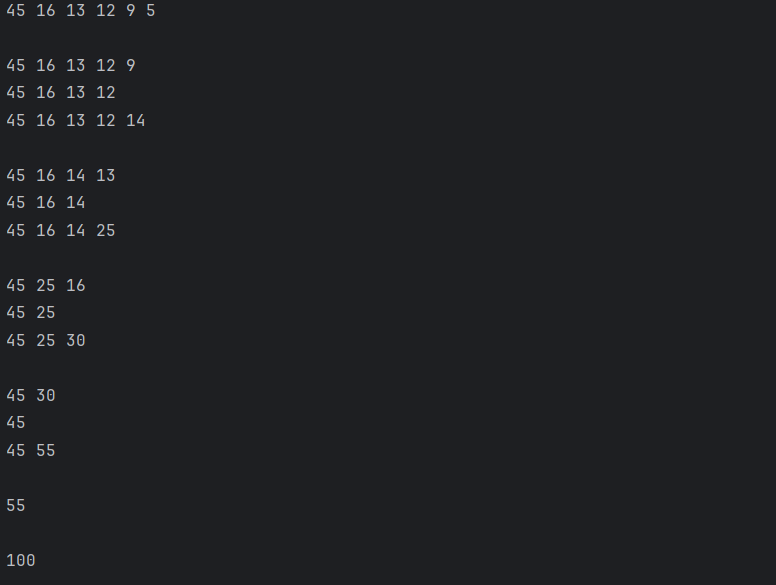
d 101

a 1100

b 1101

e 111

***Допълнително обяснение – как се променя нашия масив след всяка операция***

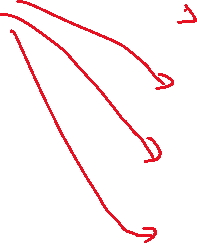
******

Масив след добавяне на комбиниран елемнент

Масив след премахване на най – малък елемент (2)

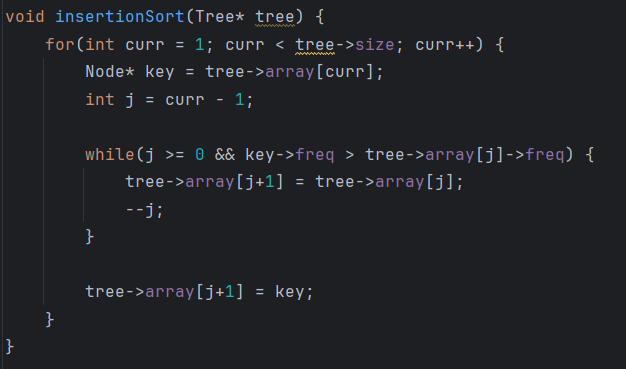
Масив след премахване на най – малък елемент (1)

Първоначален масив



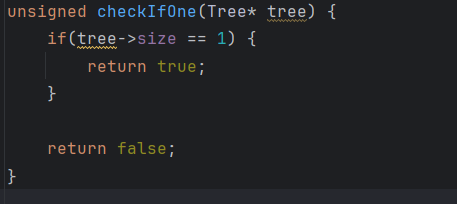
1. Методи използвани за построване на дървото
2. **Сортиращ алгоритъм**

За сортиране на нашият масив след всяко завъртане на цикъла, използваме Insertion Sort. Този алгоритъм е перфектен при малки големини на масива (максималната големина в нашият случай е 26 – броят на буквите в английската азбука)



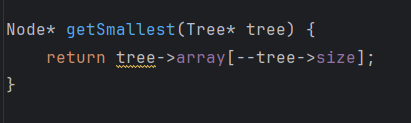
1. **Метод, проверяващ дали масива има само един елемент**

Както вече разбрахме, колкото повече се смалява масива ни, толкова по – голямо става дървото. Това е така, защото дървото е нещо, което ние не можем да видим реално. Дървото ни помага да си представим пътя от **root** елемента (в примера, **combined** със честота **100**) до всеки краен елемент. Компилаторът ни не знае какво е дърво, той открива пътя към крайния елемент чрез **left** и **right** на **combined** елементите в дървото.

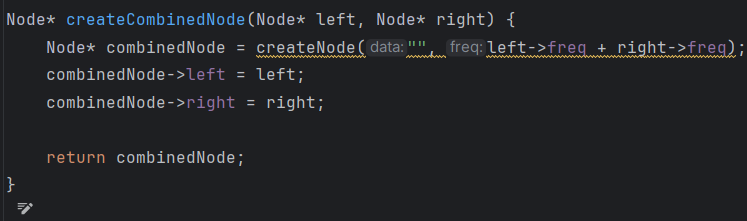


1. **Метод, връщащ последния елемент**

Този метод, връща последния елемент от масива и намалява **size**. По този начин, ние реално не трием този последен елемент, а просто го казваме на компилатора да не го взема в предвид повече.

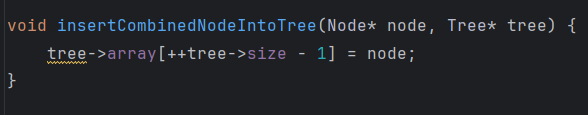


1. **Метод, създаващ комбиниран елемент**

****

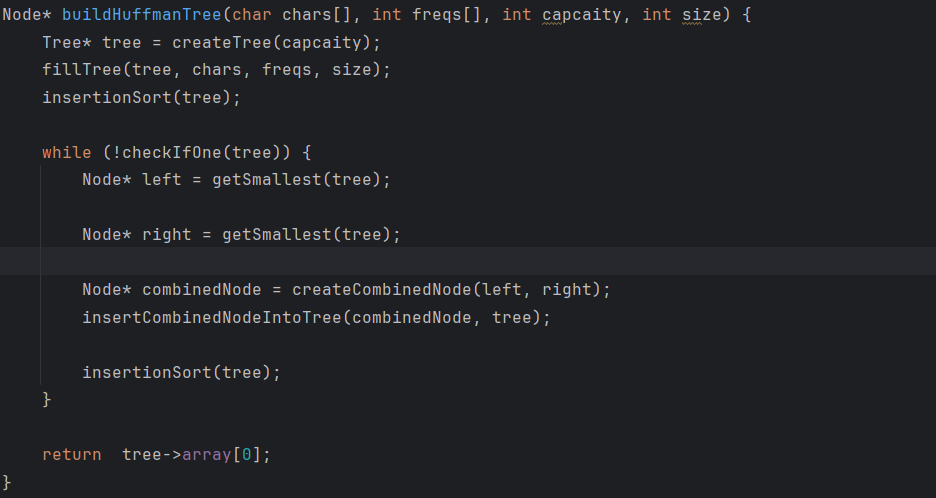
1. **Метод, поставящ комбинирания елемнт на последното място**

Подобно на метода за изтриване, този път увеличаваме големината на масива и поставяме комбинирания елемент на последното място

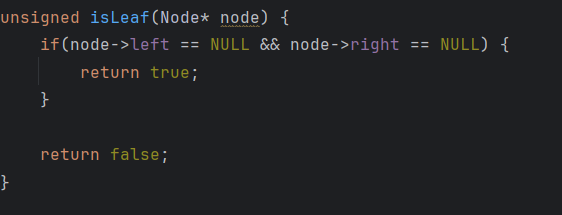
****

1. **Главен метод, строящ нашето дърво**

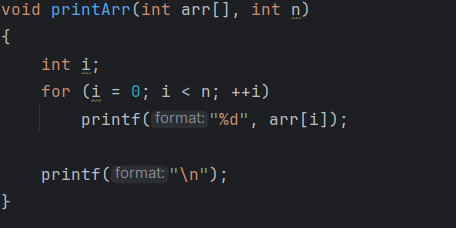
Методът **buildHuffmanTree,** използва всички методи, за ккоито говорихме досега и връща единствения останал елемент в масива. Този последн елемент представлява и нашето завършено дърво.



1. Методи за криптиране на символите и показванто им
2. **Метод за проверка дали е достигат краен елемент**



1. **Метод за принтиране на криптирания символ**



1. **Метод за достигане до краен символ и криптиране**

Това е най – интересния метод в целия код по мое мнение. **printHoffmanTree** е рекурсивен и започва винаги от **root** елемента. Той работи по следния начин:



При първото извикване от **main** метода, той приема:

* Празен масив **arr[]** – Той пази криптирания код след всяка операция
* **Top = 0 –** служи като индексатор за масива **arr**. Показва колко надълбоко сме влезли в дървото
* **rootNode** – Корена на дървото(елемента най – отгоре). При всяко последващо рекусивно извикване на функцията, **rootNode** показва елемента от дървото до който сме достигнали.

При всяко извикана на функцията, се търси винаги пътя наляво.

Ако **left** на текущия елемент не е NULL, функцията се вика рекурсивно с:

* rootNode->left – левия елемент на текущия ни елемент
* arr – криптирания масив, вече съдържащ 1 на текущата си позиция
* top + 1 – индексът на следващата позиция в arr

Ако **left** на текущият елемент е NULL, се тръгва надясно.

Прави се същата проверка и ако **right != NULL** рекурсивно се вика функцията с **rootNode->right, arr и top + 1**

Ако и **right == NULL**, значи сме достигнали до краен елемент. Принтираме стойноността на arr и рекурсивната функция продължава да обхожда останалите елементи.

Функцията достига своя край при принтиране на последния краен елемент.

1. Представяне на алгоритъма чрез псевдо код

*Algorithm Huffman (c)*

*{*

*n= |c|*

*Q = c*

*for i<-1 to n-1*

*do*

*{*

*temp <- get node ()*

*left (temp] Get\_min (Q) right [temp] Get Min (Q)*

*a = left [templ b = right [temp]*

*F [temp]<- f[a] + [b]*

*insert (Q, temp)*

*}*

*return Get\_min (0)*

*}*

1. Приложение на кода на Хуфман

Алгоритъма на Хуфман или код на Хуфман е бил широко изпползван в миналото за изпращане на факсове и работа с факс маашини. В наши дни има приложения при различни файлове като JPEG, PNG, и MP3, тъй като там има по – добра прецизност от Prefix code алгоритъма.

Приложенията на кода на Хуфман са безкрайни като той е създаден да криптира големи низове от символи, които се повтарят много често