**ВОПРОСЫ СИАОД**

**1. Типы данных, абстрактные типы данных и структуры данных.**

**Тип** данных – множество значений, которые может принимать переменная. Тип данных присваивается переменной при ее объявлении или инициализации. Основные типы данных в C++: **int** - целочисленный тип данных, **float** - тип данных с плавающей запятой, **double** - тип данных с плавающей запятой двойной точности, **char** - символьный тип данных, **bool** - логический тип данных. Процесс проверки и накладывания ограничений на типы используемых данных называется типизацией программных данных. Различают следующие **виды типизации**: **Статическая** - контроль типов осуществляется при компиляции. **Динамическая** - контроль типов осуществляется во время выполнения. Си поддерживает статическую типизацию - все используемые в программе данные должны быть указаны перед ее компиляцией. **Абстрактный** тип - математическая модель + различные операторы, определенные в рамках модели. Алгоритм разрабатывается в терминах абстрактных типов, а потом реализуется на конкретном ЯП. Для представления абстрактного типа используются **структуры данных**, которые представляют собой набор переменных различных типов, объединенных особым образом. Выбор структуры данных влияет на производительность программы.

**2. Классификация структур данных. Понятия физической и логической, простой и интегрированной структур.**

Под **структурой данных** в общем случае понимают множество элементов данных и множество связей между ними. Каждую структуру данных характеризуют **логическим** и **физическим** представлениями. Понятие «**физическая структура данных**» отражает способ физического представления данных в памяти машины и называется иначе структурой хранения, внутренней структурой или структурой памяти. Рассмотрение структуры данных без учета ее представления в машинной памяти называется абстрактной или **логической** структурой.

Различают **простые** структуры и **интегрированные**. **Простыми** называются структуры, которые не могут быть разделены на составные части, большие, чем биты (числовые, символьные, логические, перечисление, интервал, указатели). **Интегрированными** называются структуры данных, составными частями которых являются другие структуры данных. Различают **несвязные** структуры (векторы, массивы, строки, стеки, очереди) и **связные** структуры (связные списки). По признаку изменчивости различают структуры **статические**, **полустатические**, **динамические**. Базовые структуры данных, статические, полустатические и динамические характерны для оперативной памяти и часто называются **оперативными структурами**. **Файловые структуры** соответствуют структурам данных для внешней памяти.

**3. Связные и несвязные структуры данных, привести примеры. Статические, полустатические и динамические структуры, привести классификацию.**

По наличию связей между элементами данных: **несвязные** и **связные**. **Несвязные** характеризуются отсутствием связей между элементами структуры (массивы, строки, стеки, очереди). **Связные** характеризуются наличием связи (связные списки).

По изменчивости: **статические**, **динамические**. Изменчивость, то есть изменение числа элементов и/или связей между элементами структуры. **Статические** (массивы, множества, записи, таблицы). **Полустатические** (стеки, очереди, деки, строки). **Динамические** (линейные и разветвленные связные списки, графы, деревья). Для хранения информации во внешней памяти используют **файловые структуры** (последовательного доступа, прямого доступа, комбинированного доступа).

**4. Виды памяти. Ссылочный тип данных, его объявление и ситуации применения.**

Существует 3 типа памяти: статический, автоматический и динамический.

Статический — выделение памяти до начала исполнения программы. Такая память доступна на протяжении всего времени выполнения программы

Автоматический, также известный как «размещение на стеке», — самый основной, автоматически выделяет аргументы и локальные переменные функции, а также прочую метаинформацию при вызове функции и освобождает память при выходе из неё.

Динамическая — выделение памяти из ОС по требованию приложения.

Самая важная особенность ссылочных типов данных состоит в том, что они передаются не по значению, а по ссылке. Что это значит?

Ссылочные типы данных не являются примитивными и их размер не фиксирован и может быть произвольным, кроме того они хранятся не [“в переменной”.] на участке памяти переменной, а в совершенно другом месте памяти компьютера. Ссылочными типами, например, являются массивы. В объектно-ориентированных языках программирования – это экземпляры классов, коллекции и т.п.

**5. Операции над указателями. Привести примеры.**

Указатель – переменная, значением которой является адрес ячейки памяти. Указатель ссылается на блок данных из области памяти, причём на самое его начало. Указатель может ссылаться на переменную или функцию. . Для этого нужно знать адрес переменной или функции.

Чтобы узнать адрес конкретной переменной в существует унарная операция взятия адреса &. Такая операция извлекает адрес объявленных переменных, для того, чтобы его присвоить указателю.

int \*ptrvar; // объявление указателя

ptrvar = &var; // инициализация указателя

Указатели могут ссылаться на другие указатели. При этом в ячейках памяти, на которые будут ссылаться первые указатели, бу содержаться не значения, а адреса вторых указателей. Число символов \* при объявлении указателя показывает порядок указателя. Чтобы получить доступ к значению, на которое ссылается указатель его необходимо разыменовывать соответствующее количество раз.

int var = 1; // инициализация переменной var числом

int \*ptrvar = &var; // указатель на переменную var

int \*\*ptr\_ptrvar = &ptrvar;

// указатель на указатель на переменную var

int \*\*\*ptr\_ptr\_ptrvar = &ptr\_ptrvar;

(по значению указателя третьего порядка получить адрес указателя второго порядка; по значению указателя второго порядка получить адрес указателя первого порядка; по значению указателя первого порядка получить адрес переменной; по адресу переменной получить доступ к её значению).

Указатели могут ссылаться на функции. Имя функции, как и имя массива само по себе является указателем, то есть содержит адрес входа.

int nod(int, int ); // прототип указываемой функции

int main(int argc, char\* argv[])

{int (\*ptrnod)(int, int); // объявление указателя на функцию

ptrnod=nod; // присваиваем адрес функции указателю ptrnod

cout << "NOD = " << ptrnod(a, b) << endl; // обращаемся к функции через указатель

return 0;}

Основными операциями: присваивание, получение адреса, выборка. Присваивание является двухместной операцией, оба операнда которой - указатели. Как и для других типов, операция присваивания копирует значение одного указателя в другой, в результате оба указателя будут содержать один и тот же адрес памяти. Если оба указателя, участвующие в операции присваивания типизированные, то оба они должны указывать на объекты одного и того же типа.

Получение адреса - одноместная, ее операнд может иметь любой тип, результатом является типизированный (в соответствии с типом операнда) указатель, содержащий адрес объекта-операнда.

Выборка - одноместная, ее операндом является типизированный (обязательно!) указатель, результат - данные, выбранные из памяти по адресу, заданному операндом. Тип результата определяется типом указателя-операнда.

**6. Тип данных «запись». Его назначение, объявление, использование записи без вариантной части.**

Тип данных Запись (Record) используется в тех случаях, когда необходимо обрабатывать структурированные данные, которые описывают несколько различных свойств объекта.

Например, нам надо использовать следующие данные про наших друзей:

1. Фамилия

2. Имя

3. Адрес

4. Телефон

Эти данные имеют разный тип. Но из них можно составить структурированный тип данных – запись.

Описание типа данных Record

type имя записи = record

имя поля 1 : тип поля1;

- - имя поля n : тип поля n ;

end;

Например:

Структура Друзья

Фамилия

Имя

Адрес

Телефон

type friends = record

[ 12 ]

[ 12 ]

[ 25 ]

[9]

Fam : string [ 12 ];

Name : string [ 12 ];

Adress : string [ 25 ];

Telef : string [ 9 ];

end;

Составные имена полей

С полями, входящими в запись, можно выполнять те же действия, что и с обычными переменными соответствующего типа.

Для обращения к полям записи используют составные имена, части которых разделены точкой: имя записи.имя поля

Friends.Fam - фамилия друга

Friends.Telef - телефон друга

Составные имена могут участвовать в выражениях как обычные переменные:

Friends.Telef:=‘123-45-67’;

**7. Тип данных «запись». Его назначение, объявление, использование записи с вариантной частью. (6 вопрос)**

**8. Использование в записях оператора присоединения. Записи-константы.**

Использование команды присоединения With

Составные имена довольно громоздки. Чтобы иметь возможность обращаться непосредственно к самому пою в записи, используют команду With

Например:

With имя записи do

begin

действия с полями

end;

With drug do

begin

writeln ( ‘фамилия’);

readln ( fam );

writeln (‘имя’);

readln ( name);

tel := ‘276-90-90’

end;

## 

**9. Объявление и представление динамической цепочки (однонаправленного списка). Алгоритм и процедура формирования цепочки (однонаправленного списка).**

Списком называется упорядоченное множество, состоящее из переменного числа элементов, к которым применимы операции включения, исключения. Список, отражающий отношения соседства между элементами, называется *линейным*.

*Длина списка* равна числу элементов, содержащихся в списке, список нулевой длины называется *пустым списком*. Списки представляют собой способ организации структуры данных, при которой элементы некоторого типа образуют цепочку. Для связывания элементов в списке используют систему указателей. В минимальном случае, любой элемент линейного списка имеет один указатель, который указывает на следующий элемент в списке или является пустым указателем, что интерпретируется как конец списка.

Структура, элементами которой служат записи с одним и тем же форматом, связанные друг с другом с помощью указателей, хранящихся в самих элементах, называют *связанным списком*. В связанном списке элементы линейно упорядочены, но порядок определяется не номерами, как в массиве, а указателями, входящими в состав элементов списка. Каждый список имеет особый элемент, называемый *указателем начала списка (головой списка)*, который обычно по содержанию отличен от остальных элементов. В поле указателя последнего элемента списка находится специальный признак NULL, свидетельствующий о конце списка.

### Однонаправленные (односвязные) списки

Наиболее простой динамической структурой является однонаправленный список, элементами которого служат объекты структурного типа.

Однонаправленный (односвязный) список – это структура данных, представляющая собой последовательность элементов, в каждом из которых хранится значение и указатель на следующий элемент списка В последнем элементе указатель на следующий элемент равен NULL.



Описание простейшего элемента такого списка выглядит следующим образом:

struct имя\_типа { информационное поле; адресное поле; };

где информационное поле – это поле любого, ранее объявленного или стандартного, типа;

адресное поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка.

Например:

struct Node {

int key;//информационное поле

Node\*next;//адресное поле

};

Информационных полей может быть несколько.

Например:

struct point {

char\*name;//информационное поле

int age;//информационное поле

point\*next;//адресное поле

};

Каждый элемент списка содержит ключ, который идентифицирует этот элемент. Ключ обычно бывает либо целым числом, либо строкой.

Основными операциями, осуществляемыми с однонаправленными списками, являются:

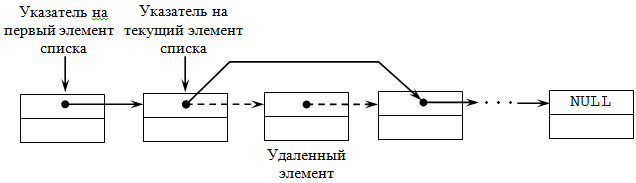
* создание списка;
* печать (просмотр) списка;
* вставка элемента в список;
* удаление элемента из списка;
* поиск элемента в списке
* проверка пустоты списка;
* удаление списка.

Особое внимание следует обратить на то, что при выполнении любых операций с линейным однонаправленным списком необходимо обеспечивать позиционирование какого-либо указателя на первый элемент. В противном случае часть или весь список будет недоступен.

**10. Объявление и представление динамической цепочки (однонаправленного списка). Алгоритм и процедура удаления звена цепочки (однонаправленного списка).**

Из динамических структур можно удалять элементы, так как для этого достаточно изменить значения адресных полей. Операция удаления элемента однонаправленного списка осуществляет удаление элемента, на который установлен указатель текущего элемента. После удаления указатель текущего элемента устанавливается на предшествующий элемент списка или на новое начало списка, если удаляется первый.

Алгоритмы удаления первого и последующих элементов списка отличаются друг от друга. Поэтому в функции, реализующей данную операцию, осуществляется проверка, какой элемент удаляется. Далее реализуется соответствующий алгоритм удаления

****

Single\_List\* Delete\_Item\_Single\_List(Single\_List\* Head,

int Number){

Single\_List \*ptr;//вспомогательный указатель

Single\_List \*Current = Head;

for (int i = 1; i < Number && Current != NULL; i++)

Current = Current->Next;

if (Current != NULL){//проверка на корректность

if (Current == Head){//удаляем первый элемент

Head = Head->Next;

delete(Current);

Current = Head;

}

else {//удаляем непервый элемент

ptr = Head;

while (ptr->Next != Current)

ptr = ptr->Next;

ptr->Next = Current->Next;

delete(Current);

Current=ptr;

}

}

return Head;

}

**11. Объявление и представление динамической цепочки (однонаправленного списка). Алгоритм и процедура вставки звена в цепочку после заданного.**

В динамические структуры легко добавлять элементы, так как для этого достаточно изменить значения адресных полей. Вставка первого и последующих элементов списка отличаются друг от друга. Поэтому в функции, реализующей данную операцию, сначала осуществляется проверка, на какое место вставляется элемент. Далее реализуется соответствующий алгоритм добавления

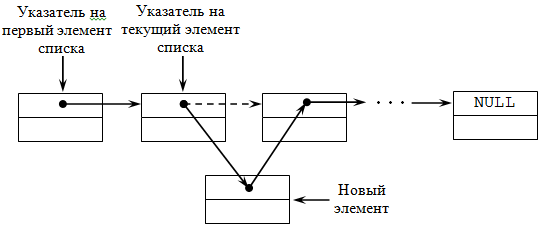


Рис. 29.2. Вставка элемента в однонаправленный список

/\*вставка элемента с заданным номером в однонаправленный список\*/

Single\_List\* Insert\_Item\_Single\_List(Single\_List\* Head,

int Number, int DataItem){

Number--;

Single\_List \*NewItem=new(Single\_List);

NewItem->Data=DataItem;

NewItem->Next = NULL;

if (Head == NULL) {//список пуст

Head = NewItem;//создаем первый элемент списка

}

else {//список не пуст

Single\_List \*Current=Head;

for(int i=1; i < Number && Current->Next!=NULL; i++)

Current=Current->Next;

if (Number == 0){

//вставляем новый элемент на первое место

NewItem->Next = Head;

Head = NewItem;

}

else {//вставляем новый элемент на непервое место

if (Current->Next != NULL)

NewItem->Next = Current->Next;

Current->Next = NewItem;

}

}

return Head;

}

**12. Структура звена двунаправленного списка. Два вида двунаправленных списков. Алгоритм и процедура вставки элемента в двунаправленный список.**

Двунаправленные (двусвязные) списки

Для ускорения многих операций целесообразно применять переходы между элементами списка в обоих направлениях. Это реализуется с помощью двунаправленных списков, которые являются сложной динамической структурой.

Двунаправленный (двусвязный) список – это структура данных, состоящая из последовательности элементов, каждый из которых содержит информационную часть и два указателя на соседние элементы ( [рис. 29.4](https://intuit.ru/studies/professional_skill_improvements/2056/courses/504/lecture/11456?page=3#image.29.4)). При этом два соседних элемента должны содержать взаимные ссылки друг на друга.

В таком списке каждый элемент (кроме первого и последнего) связан с предыдущим и следующим за ним элементами. Каждый элемент двунаправленного списка имеет два поля с указателями: одно поле содержит ссылку на следующий элемент, другое поле – ссылку на предыдущий элемент и третье поле – информационное. Наличие ссылок на следующее звено и на предыдущее позволяет двигаться по списку от каждого звена в любом направлении: от звена к концу списка или от звена к началу списка, поэтому такой список называют двунаправленным.

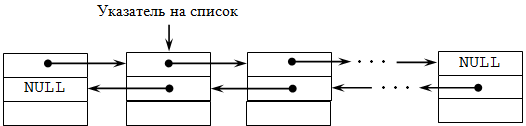


Рис. 29.4. Двунаправленный список

Описание простейшего элемента такого списка выглядит следующим образом:

struct имя\_типа {

информационное поле;

адресное поле 1;

адресное поле 2;

};

где информационное поле – это поле любого, ранее объявленного или стандартного, типа;

адресное поле 1 – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка ;

адресное поле 2 – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес предыдущего элемента списка.

В динамические структуры легко добавлять элементы, так как для этого достаточно изменить значения адресных полей. Операция вставки реализовывается аналогично функции вставки для однонаправленного списка, только с учетом особенностей двунаправленного списка ( [рис. 29.5](https://intuit.ru/studies/courses/648/504/lecture/11456?page=3#image.29.5)).

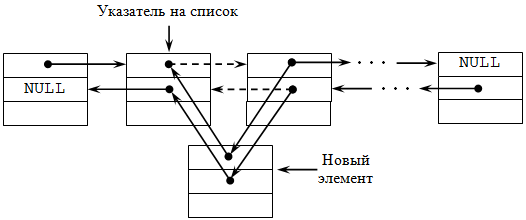


Рис. 29.5. Добавление элемента в двунаправленный список

//вставка элемента с заданным номером в двунаправленный список

Double\_List\* Insert\_Item\_Double\_List(Double\_List\* Head,

int Number, int DataItem){

Number--;

Double\_List \*NewItem=new(Double\_List);

NewItem->Data=DataItem;

NewItem->Prior=NULL;

NewItem->Next = NULL;

if (Head == NULL) {//список пуст

Head = NewItem;

}

else {//список не пуст

Double\_List \*Current=Head;

for(int i=1; i < Number && Current->Next!=NULL; i++)

Current=Current->Next;

if (Number == 0){

//вставляем новый элемент на первое место

NewItem->Next = Head;

Head->Prior = NewItem;

Head = NewItem;

}

else {//вставляем новый элемент на непервое место

if (Current->Next != NULL) Current->Next->Prior = NewItem;

NewItem->Next = Current->Next;

Current->Next = NewItem;

NewItem->Prior = Current;

Current = NewItem;

}

}

return Head;

}

**13. Структура звена двунаправленного списка. Алгоритм и процедура удаления элемента из двунаправленного списка.**

Из динамических структур можно удалять элементы, так как для этого достаточно изменить значения адресных полей. Операция удаления элемента из двунаправленного списка осуществляется во многом аналогично удалению из однонаправленного списка ( [рис. 29.6](https://intuit.ru/studies/courses/648/504/lecture/11456?page=4#image.29.6)).

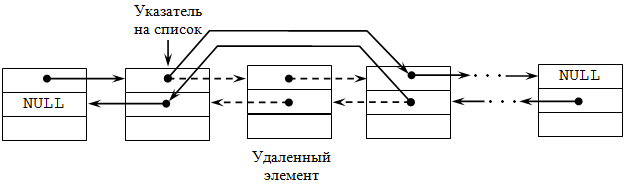


Рис. 29.6. Удаление элемента из двунаправленного списка

/\*удаление элемента с заданным номером из двунаправленного списка\*/

Double\_List\* Delete\_Item\_Double\_List(Double\_List\* Head,

int Number){

Double\_List \*ptr;//вспомогательный указатель

Double\_List \*Current = Head;

for (int i = 1; i < Number && Current != NULL; i++)

Current = Current->Next;

if (Current != NULL){//проверка на корректность

if (Current->Prior == NULL){//удаляем первый элемент

Head = Head->Next;

delete(Current);

Head->Prior = NULL;

Current = Head;

}

else {//удаляем непервый элемент

if (Current->Next == NULL) {

//удаляем последний элемент

Current->Prior->Next = NULL;

delete(Current);

Current = Head;

}

else {//удаляем непервый и непоследний элемент

ptr = Current->Next;

Current->Prior->Next =Current->Next;

Current->Next->Prior =Current->Prior;

delete(Current);

Current = ptr;

}

}

}

return Head;

}

**14. Структура звена двунаправленного списка. Алгоритм и процедура поиска элемента в двунаправленном списке.**

Операция поиска элемента в двунаправленном списке реализуется абсолютно аналогично соответствующей функции для однонаправленного списка. Поиск элемента в двунаправленном списке можно вести:

а) просматривая элементы от начала к концу списка;

б) просматривая элементы от конца списка к началу;

в) просматривая список в обоих направлениях одновременно: от начала к середине списка и от конца к середине (учитывая, что элементов в списке может быть четное или нечетное количество).

//поиск элемента в двунаправленном списке

bool Find\_Item\_Double\_List(Double\_List\* Head,

int DataItem){

Double\_List \*ptr; //вспомогательный указатель

ptr = Head;

while (ptr != NULL){//пока не конец списка

if (DataItem == ptr->Data) return true;

else ptr = ptr->Next;

}

return false;

}

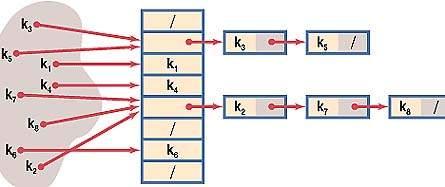
**15. Назначение хеширования данных. Открытое хеширование. Привести пример организации данных.**

Для сокращения времени доступа к данным в таблицах используется так называемое случайное упорядочивание или хеширование.

При открытом хешировании таблица рассматривается как массив связанных списков. Каждый такой список называется блоком и содержит записи, отображаемые хеш-функцией в один и тот же табличный адрес. Эта стратегия разрешения коллизий называется методом цепочек.

Если таблица является массивом связанных списков, то элемент данных просто вставляется в соответствующий список в качестве нового узла. Чтобы обнаружить элемент данных, нужно применить хеш-функцию для определения нужного связанного списка и выполнить там последовательный поиск.

Каждый новый элемент данных вставляется в хвост соответствующего связанного списка.



+

В общем случае метод цепочек быстрее открытой адресации, так как просматривает только те ключи, которые попадают в один и тот же табличный адрес. Кроме того, открытая адресация предполагает наличие таблицы фиксированного размера, в то время как в методе цепочек элементы таблицы создаются динамически, а длина списка ограничена лишь количеством памяти. Основным недостатком метода цепочек являются дополнительные затраты памяти на поля указателей. В общем случае динамическая структура метода цепочек более предпочтительна для хеширования.

**16. Назначение хеширования данных. Закрытое хеширование. Привести пример организации данных.**

Открытое хеширование позволял хранить сколь угодно много элементов, а при закрытом хешировании их количество ограничено размером хеш-таблицы. Закрытое не требует каких-либо дополнительных структур данных. В ячейках таблицы хранятся не указатели, а элементы исходного массива, доступ к каждому из которых осуществляется по хеш-значению ключа, при этом одна ячейка может содержать только один элемент. При закрытом хешировании в таблице сегментов хранятся сами элементы множеств, а не заголовки их списков. При закрытом хешировании применяется методика повторного хеширования, т.е. если мы пытаемся поместить элемент Х в сегмент с номером Н(х), который занят другим элементом (коллизия), то в соответствии с методом повторного хеширования выбирается последовательность других номеров сегментов Н1(х), Н2(х), …, куда можно поместить элемент Х. Каждый из них проверяется, пока не будет найдено свободное местоположение (сегмент), если свободных сегментов нет, то таблица заполняется, и элемент Х вставить нельзя. Пример:

B=8

a, b, c, d

h(a)=3, h(b)=0, h(c)=4, h(d)=3

h1 (x)= (h(x)+i)mod B

**17. Разрешение коллизий в случае закрытого хеширования.**

При заполнении таблицы могут возникать коллизии, для борьбы с которыми разработаны специальные методы, которые в основном сводятся к методам "цепочек" и "открытой адресации". Ключи, выдающие одинаковые адреса в таблице, называются ключи-синонимы.

В методе цепочек для разрешения коллизий во все записи вводятся указатели, используемые для организации списков – "цепочек переполнения". В случае возникновения коллизии при заполнении таблицы в список для требуемого адреса хеш-таблицы добавляется еще один элемент.

Хранит все элементы с данным хэшом данной таблицы. Если есть коллизия то мы добавим новый лемент в конец спска. И при поиске найдя хэш пробегаем по содержимаму.

Поиск в хеш-таблице с цепочками переполнения осуществляется следующим образом. Сначала вычисляется адрес по значению ключа. Затем осуществляется последовательный поиск в списке, связанном с вычисленным адресом. Процедура удаления из таблицы сводится к поиску элемента и его удалению из цепочки переполнения.

Метод открытой адресации состоит в том, чтобы, пользуясь каким-либо алгоритмом, обеспечивающим перебор элементов таблицы, просматривать их в поисках свободного места для новой записи.

Различают:

а) Линейное пробирование

б) Квадратичное пробирование

в) Двойное хеширование

Линейное пробирование сводится к последовательному перебору элементов таблицы с некоторым фиксированным шагом

Квадратичное пробирования отличается от линейного тем, что шаг перебора элементов нелинейно зависит от номера попытки найти свободный элемент

Интервал между ячейками фиксирован, как при линейном пробировании, но, в отличие от него, размер интервала вычисляется второй, вспомогательной хеш-функцией, а значит может быть различным для разных ключей.

Значения этой хеш-функции должны быть ненулевыми и взаимно-простыми с размером хеш-таблицы, что проще всего достичь, взяв простое число в качестве размера, и потребовав, чтобы вспомогательная хеш-функция принимала значения от 1 до N — 1.

**18. Алгоритмы работы с хеш-таблицами методами открытой адресации.**

Открытая адресация

Как и отдельная цепочка, открытая адресация является методом обработки коллизий. В открытой адресации все элементы хранятся в самой хеш-таблице. Таким образом, в любой момент размер таблицы должен быть больше или равен общему количеству ключей (обратите внимание, что мы можем увеличить размер таблицы, копируя старые данные, если это необходимо).

Вставка (k): продолжайте зондирование, пока не найдете пустую щель. Найдя пустой слот, вставьте k.

Поиск (k): Продолжайте исследование, пока ключ слота не станет равным k или не будет достигнут пустой слот.

Удалить (k): *операция удаления интересна* . Если мы просто удалим ключ, поиск может закончиться неудачей. Поэтому слоты удаленных ключей специально помечаются как «удаленные».

Вставка может вставить элемент в удаленный слот, но поиск не останавливается на удаленном слоте.

Открытая адресация осуществляется следующими способами:

*a) Линейное зондирование: при* линейном зондировании мы линейно зондируем следующий слот. Например, типичный зазор между двумя зондами равен 1, как показано в примере ниже.

пусть hash (x) будет индексом слота, вычисленным с использованием хеш-функции, а S будет размером таблицы

If slot hash(x) % S is full, then we try (hash(x) + 1) % S

If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S

If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S

..................................................

..................................................

Давайте рассмотрим простую хеш-функцию как «ключ мод 7», а последовательность ключей — 50, 700, 76, 85, 92, 73, 101.

Кластеризация: Основная проблема с линейным зондированием — это кластеризация, многие последовательные элементы образуют группы, и начинается поиск свободного слота или поиск элемента.

*б) Квадратичное исследование.* Мы ищем 2 -й слот в i-й итерации.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1\*1) % S

If (hash(x) + 1\*1) % S is also full, then we try (hash(x) + 2\*2) % S

If (hash(x) + 2\*2) % S is also full, then we try (hash(x) + 3\*3) % S

..................................................

..................................................

c) [Двойное хеширование.](http://espressocode.top/double-hashing/) Мы используем другую хеш-функцию hash2 (x) и ищем слот i \* hash2 (x) в i-ом повороте.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1\*hash2(x)) % S

If (hash(x) + 1\*hash2(x)) % S is also full, then we try (hash(x) + 2\*hash2(x)) % S

If (hash(x) + 2\*hash2(x)) % S is also full, then we try (hash(x) + 3\*hash2(x)) % S

..................................................

..................................................

**19. Абстрактный тип данных «очередь». Алгоритм и процедура занесения элемента в очередь.**

Рассмотрим алгоритм добавления только для второго элемента.

1. Ввод информации для текущего (второго) элемента – значение *i* .

2. Захватываем память под текущий элемент:

t = (Spis\*) malloc (sizeof(Spis)); или t = new Spis;

3. Формируем информационную часть (обозначим *i*2):

t -> info = i;

4. В адресную часть созданного элемента (текущего) заносим *NULL*, т.к. этот элемент становится последним:

t -> Next = NULL;

5. Элемент добавляется в конец очереди, поэтому в адресную часть бывшего последнего элемента *end* заносим адрес созданного:

end -> Next = t;

бывший последний элемент становится предпоследним.

6. Переставляем указатель последнего элемента на добавленный:

end = t;

В результате получим

Для добавления в очередь любого количества элементов организуется цикл, включающий пункты 1– 6 рассмотренного алгоритма. Завершение цикла реализуется в зависимости от поставленной задачи.

**20. Абстрактный тип данных «очередь». Алгоритм и процедура удаления элемента из очереди.**

Очередь представляет собой линейный список данных, доступ к которому осуществляется по принципу "первый вошел, первый вышел" /иногда сокращенно его называют методом доступа FIFO/. Элемент, который был первым поставлен в очередь, будет первым получен при поиске. Элемент, поставленный в очередь вторым, при поиске будет получен также вторым и т.д. Этот способ является единственным при постановке элементов в очередь и при поиске элементов в очереди. Применение очереди не позволяет делать прямой доступ к любому конкретному элементу. Чтобы удалить элемент из очереди, то требуется доставать элементы по очереди в другую очередь, а нужный элемент не сохранить

**21. Абстрактный тип данных «стек». Алгоритм и процедура занесения элемента в стек с помощью указателей.**

Организация стека в определенном смысле противоположна организации очереди, поскольку здесь используется доступ по принципу "последней вошел, первый вышел" /такой метод доступа иногда называют методом LIFO/. Представим себе стопку тарелок. Нижняя тарелка из этой стопки будет использована последней, а верхняя тарелка /которая была установлена в стопку последней/ будет использована первой. Стеки широко используются в системном программном обеспечении, включая компиляторы и интерпретаторы.

Исторически сложилось так, что две основные операции для стека - поместить в стек и выбрать из стека - получили название соответственно "затолкнуть" и "вытолкнуть". Поэтому для реализации стека необходимо создать две функции: "posh" /затолкнуть/, которая помещает элемент в вершину стека, и "pop" /вытолкнуть/, которая выбирает из вершины стека значение.

**22. Абстрактный тип данных «стек». Алгоритм и процедура удаления элемента из стека с помощью указателей.**

Спец. тип списка(LIFO), в котором все операции выполняются на одном конце, называемом вершиной. Значение указателя стека – ссылка на вершину. Каждый эл-т содержит ссылку на следующий.

Описание:type

pt=^elem;

elem=record

data:integer;

next:pt;

end;

procedure writestack(var u:pt;dig:integer);

var

x:pt;

begin

new(x);

x^.data:=dig;

x^.next:=u;

u:=x;

end;

u-ссылка на стек;dig-записыв. значение.

procedure readstack(var u:pt; var dig:integer);

var

x:pt;

begin

dig:=u^.data;

x:=u;

u:=u^.next;

dispose(x);

end;

В результате процедуры переменной будет присвоено значение первого элемента и изменена ссылка на вершину стека.

**23.Постфиксная, префиксная, инфиксная записи представления выражений и их особенности. Привести примеры.**

3 формы записи:

1)А+В-инфиксная.

2)+АВ-префиксная(польская)

3)АВ+-постфиксная(обр. польская)

Метод польской записи создан при исследовании трансляции Яном Лукасевичем..Для преобразования в префиксн. используются приоритеты.Высший учитыв. первым. После вычислений результат-один операнд.Вычисления – слева направо, кроме возведения в степень.

Сущность польской записи – в отсутствии скобок и порядке знаков, который соответствует порядку действий слева направо. Любое выражение можно вычислить за один проход.Для преобразований использ. стек. Ранг результата должен равняться 1.

1)В стек помещается символ пустого стека.

2)Приоритет входного символа сравнивается с пиоритетом верхнего эл-та стека.

3)Если приоритет входного >, то символ заносится в стек, иначе верхний выталкивается из стека. Сравнение происходит снова.

При каждом добавлении в префиксн. запись модифицир. ранг.

+,- 1

\*,/ 2

a..z 3

Дно стека 0

При преобразовании инфиксного в польское порядок переменных не меняется,порядок операторов меняется отн-но приоритета.

**24. Использование стека операций для перевода выражений из инфиксной в постфиксную запись. Привести алгоритм.**

До сих пор мы использовали специальные методы для преобразования между инфиксными выражениями и эквивалентными им префиксной и постфикской записями. Как вы можете ожидать, существуют алгоритмические способы выполнения таких преобразований, позволяющие корректно трансформировать любое выражение любой сложности.

Первой из рассматриваемых нами техник будет использование идеи полной расстановки скобок в выражении, рассмотренной нами ранее. Напомним, что A + B \* C можно записать как (A + (B \* C)), чтобы явно показать приоритет умножения перед сложением. Однако, при более близком рассмотрении вы увидите, что каждая пара скобок также отмечает начало и конец пары операндов с соответствующим оператором по середине.

Взгляните на правую скобку в подвыражении (B \* C) выше. Если мы передвинем символ умножения с его позиции и удалим соответствующую левую скобку, получив B C \*, то произойдёт конвертирование подвыражение в постфиксную нотацию. Если оператор сложения тоже передвинуть к соответствующей правой скобке и удалить связанную с ним левую скобку, то результатом станет полностью постфиксное выражение (см. [рисунок 6](https://aliev.me/runestone/BasicDS/InfixPrefixandPostfixExpressions.html#fig-moveright)).

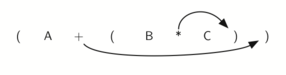


Рисунок 6: Перемещение операторов вправо для постфиксной записи

Если мы сделаем тоже самое, но вместо передвижения символа на позицию к правой скобке, сдвинем его к левой, то получим префиксную нотацию (см. [рисунок 7](https://aliev.me/runestone/BasicDS/InfixPrefixandPostfixExpressions.html#fig-moveleft)). Позиция пары скобок на самом деле является ключом к окончательной позиции заключённого между ними оператора.

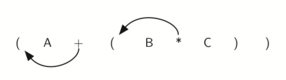


Рисунок 7: Перемещение операторов влево для префиксной записи.

Таким образом, при преобразовании выражения (неважно, насколько сложного) в префиксную или постфиксную запись для установления порядка выполнения операций используется полная расстановка скобок. Затем находящийся внутри них оператор передвигается на крайнюю левую или крайнюю правую позицию - в зависимости от того, префиксную или постфиксную запись вы хотите получить.

Вот более сложное выражение: (A + B) \* C - (D - E) \* (F + G). [Рисунок 8](https://aliev.me/runestone/BasicDS/InfixPrefixandPostfixExpressions.html#fig-complexmove) демонстрирует его преобразование в постфиксный и префиксный виды.

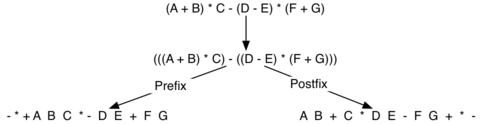


Рисунок 8: Преобразование сложного выражения к префиксной и постф

иксной записи.

**25. Использование стека операций для перевода выражений из инфиксной в префиксную запись. Привести алгоритм.**

Идея состоит в том, чтобы использовать один стек для хранения операторов, а другой — для хранения операндов. Пошаговый алгоритм это:

1. Пройдите по инфиксному выражению и проверьте, является ли данный символ оператором или операндом.
2. Если это операнд, поместите его в стек операндов.
3. Если это оператор, то проверьте, является ли приоритет текущего оператора больше или меньше или равен оператору на вершине стека. Если приоритет больше, поместите оператор в стек оператора. В противном случае извлеките два операнда из стека операндов, выдвиньте оператор из стека операторов и вставьте оператор строки + операнд1 + операнд 2 в стек операндов. Продолжайте извлекать из обоих стеков и помещать результат в стек операндов, пока приоритет текущего оператора не станет меньше или равен оператору в верхней части стека операторов.
4. Если текущим символом является '(', то поместите его в стек операторов.
5. Если текущим символом является ')', то проверьте, является ли вершина стека операторов открывающей скобкой или нет. Если нет, извлеките два операнда из стека операндов, вытолкните оператор из стека операторов и вставьте строковый оператор + операнд1 + операнд 2 в стек операндов. Продолжайте выталкивать из обоих стеков и помещать результат в стек операндов, пока вершина стека операторов не станет открывающей скобкой.
6. Окончательное префиксное выражение присутствует в верхней части стека операндов.

**26. Правило вычисления выражения в постфиксной записи.**

Постфиксная запись представляет собой такую запись арифметического выражения, в которой сначала записываются операнды, а затем – знак операции. Например, для выражения a + b \* c постфиксная запись будет a b c \* +. Здесь операндами операции \* будут b и c (два ближайших операнда), а операндами операции + будут а и составной операнд b c \*. Эта запись удобна тем, что она не требует скобок. Например, для выражения (a + b) \* c постфиксная запись будет a b + c \*. В этой записи не требуется ставить скобки для того, чтобы изменить порядок вычисления, зависящий от приоритета операций, как в исходном выражении.

Алгоритм перевода в постфиксную запись обрабатывает исходный массив лексем и строит новый массив из тех же лексем, расположенных в другом порядке. Кроме того, необходим еще стек – аналогичный массив, используемый для временного хранения операций.

Алгоритм перевода выражения в постфиксную запись следующий.

1. Константы и переменные кладутся в формируемую запись в порядке их появления в исходном массиве.
2. При появлении операции в исходном массиве:
   1. если в стеке нет операций или верхним элементом стека является открывающая скобка, операции кладётся в стек;
   2. если новая операции имеет больший\* приоритет, чем верхняя операции в стеке, то новая операции кладётся в стек;
   3. если новая операция имеет меньший или равный приоритет, чем верхняя операции в стеке, то операции, находящиеся в стеке, до ближайшей открывающей скобки или до операции с приоритетом меньшим, чем у новой операции, перекладываются в формируемую запись, а новая операции кладётся в стек.
3. Открывающая скобка кладётся в стек.
4. Закрывающая скобка выталкивает из стека в формируемую запись все операции до ближайшей открывающей скобки, открывающая скобка удаляется из стека.
5. После того, как мы добрались до конца исходного выражения, операции, оставшиеся в стеке, перекладываются в формируемое выражение.

**27. Формальное определение типа данных «дерево». Отношения между узлами в дереве. Понятия предок, потомок, путь, длина пути. Примеры.**

**Дерево** — структура данных, эмулирующая древовидную структуру в виде набора связанных узлов. Является связным графом, не содержащим циклы.

**Между узлами существуют связи:**

**Отношение предка** - если узел связан с узлом более близким к корню.

**Отношении потомка** - если узел связан с узлом более низкого уровня.

**Отношение близнецы** - узлы имеют общего предка.

**Предок какого-то узла** — это узел, из которого можно перейти по стрелкам в данный узел.

**Потомок какого-то узла** — это узел, в который можно перейти по стрелкам от узла-предка.

**Путь** **в** **графе** — последовательность вершин, в которой каждая вершина соединена со следующим ребром.

**Длина пути** - количество ребер, из которых состоит путь.

**28. Описание вершины дерева. Понятие бинарного дерева поиска.** Процедура вставки элемента в бинарное дерево поиска.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено и поиск прекращается.

**28. Понятие обхода дерева. Рекурсивное определение и процедура прямого обхода дерева. Привести пример.**

Обход дерева - получение всех элементов дерева.

В коде рекурсивной функции соответствующего обхода сохраняется соответствующий порядок вызовов (порядок строк кода), где вместо корня идет вызов данной рекурсивной функции.

Прямой обход - проверяем не является ли текущий узел null, показываем данные узла, обходим левое поддерево рекурсивно, обходим правое дерево рекурсивно.

NLR(node)  
{

if(node!=null) {

cww(node.value);

NLR(node.lde.right);

}

}eft);

NLR(no

**29. Понятие обхода дерева. Рекурсивное определение и процедура симметричного обхода дерева. Привести пример.**

Обход дерева - получение всех элементов дерева.

В коде рекурсивной функции соответствующего обхода сохраняется соответствующий порядок вызовов (порядок строк кода), где вместо корня идет вызов данной рекурсивной функции.

Cимметричный обход - проверяем не является ли текущий узел null, обходим левое поддерево рекурсивно,показываем данные узла, обходим правое дерево рекурсивно.

LNR(node)  
{

if(node!=null) {

LNR(node.left);

cww(node.value);

LNR(node.right);

}

**}**

**30. Понятие обхода дерева. Рекурсивное определение и процедура обратного обхода дерева. Привести пример.**

Обход дерева - получение всех элементов дерева.

В коде рекурсивной функции соответствующего обхода сохраняется соответствующий порядок вызовов (порядок строк кода), где вместо корня идет вызов данной рекурсивной функции.

Обратный обход обход - проверяем не является ли текущий узел null, обходим левое поддерево рекурсивно, обходим правое дерево рекурсивно,показываем данные узла.

LRN(node)  
{

if(node!=null) {

LRN(node.left);

LRN(node.right);

cww(node.value);

}

**31. Описание вершины дерева. Процедура поиска в дереве элемента с заданным ключом.**

Требуется найти элемент, равный *v*, в дереве. Введем понятие текущей вершины дерева *c*. Сначала в качестве *c* выберем корень дерева. Рекурсивно вызывается следующая процедура:



*Если c==NULL то ИСКОМЫЙ ЭЛЕМЕНТ В ДЕРЕВЕ НЕ СОДЕРЖИТСЯ*

*Если v==c то return c*

*Если vc то выполнить эту же процедуру для c->right*

*иначе выполнить эту же процедуру для c->left*

**

На языке С это будет выглядеть следующим образом

*STree \*Find(STree \*root, int v)*

*{*

*if(root==NULL)return NULL;*

*if(root->value==v)return root;*

*if(root->value>=v)return Find(root->right,v);*

*else return Find(root->left,v);*

*}*

или более коротко:

*STree \*Find(STree \*root, int v)*

*{*

*return root==NULL?NULL: root->value==v? root: v>=root->value?*

*Find(root->right,v): Find(root->left,v);*

*}*

**32. Описание вершины дерева. Процедура вставки в дерево элемента с заданным ключом.**

Требуется добавить в дерево вершину ***v***.

Для этого ищем лист, после которого следует вставить ***v*** и вставляем ***v*** после него.

Алгоритмом, аналогичным поиску элемента, найдем лист ***c***, после которого следует вставить элемент ***v*** и вставим его. На языке С вставка вершины в дерево может быть выполнена следующим образом:

*STree \*Insert(STree \*root, STree \*v)*

*{*

*if(v->value>=root->value)*

*return root->right==NULL ?*

*(v->back=root,v->right=v->left=NULL,root->right=v) : Insert(root->right, v);*

*else*

*return root->left==NULL ?*

*(v->back=root,v->right=v->left=NULL,root->left=v) : Insert(root->left, v);*

*}*

**33. Ситуации удаления элемента из дерева. Процедура удаления заданного элемента из дерева.**

. Операция удаления элемента из дерева поиска сложнее, чем операция включения элемента. После удаления элемента дерево поиска должно сохранить свое свойство — обеспечение поиска элементов. Действия по удалению определяются тем, в каком месте дерева находится удаляемый элемент. В процедуре удаления различают следующие ситуации.

1. Элемента с искомым ключом нет, дерево не изменяется, возврат с признаком отсутствия ключа.
2. Удаляемый элемент — лист, в родительском узле соответствующий указатель на удаляемый элемент обнуляется, память из-под элемента освобождается.
3. Удаляемый элемент только с одним поддеревом. Корректируются указатели, память освобождается.
4. Удаляемый элемент имеет два поддерева. В этом случае необходимо спуститься вдоль самой правой ветви левого поддерева до конца и заменить удаляемый элемент конечным элементом с корректировкой указателей. Память освобождается. Вместо эго можно спуститься до конца вдоль самой левой ветви правого поддерева и заменить удаляемый элемент конечным элементом, скорректировать указатели и освободить память.

Tree\* DeleteNode(Tree\* node, int val){

if(node == NULL)

return node;

if(val == node->val){

Tree\* tmp;

if(node->right == NULL)

tmp = node->left;

else {

Tree\* ptr = node->right;

if(ptr->left == NULL){

ptr->left = node->left;

tmp = ptr;

} else {

Tree\* pmin = ptr->left;

while(pmin->left != NULL){

ptr = pmin;

pmin = ptr->left;

}

ptr->left = pmin->right;

pmin->left = node->left;

pmin->right = node->right;

tmp = pmin;

}

}

delete node;

return tmp;

} else if(val < node->val)

node->left = DeleteNode(node->left, val);

else

node->right = DeleteNode(node->right, val);

return node;

}

**34. Помеченные деревья. Правила соответствия меток деревьев элементам выражений. Привести пример прямого, обратного и симметричного обходов помеченного дерева.**

**35. Алгоритм построения помеченного дерева по выражению и обходы дерева. Привести пример.**

**36. Структура узла прошитого дерева. Процедура симметричной прошивки бинарного дерева.**

**37. Структура узла прошитого дерева. Процедура обхода симметрично прошитого бинарного дерева.**

**38. Структура узла прошитого дерева. Процедура прямой прошивки бинарного дерева.**

Структура узла:

const int THREAD=0;

const int MAINLINK=1;

struct NODE{

<поля данных>;

NODE \*Left, \*Right;

};

Рассмотрим алгоритм правосторонней симметричной прошивки бинарного дерева.

1. Строится бинарное дерево. При этом поля *ltag* и *rtag* создаваемых узлов дерева остаются неопределенными, а *left* и *right* соответственно указывают на левое или правое поддеревья, либо равны *nil*. На корень построенного дерева указывает *root*.
2. Создается головной узел, *left* которого указывает на корень дерева, а *right* на сам головной узел:

*new(HEAD);*

*HEAD^.left := root;*

*HEAD^.right := HEAD;*

Информационное поле и поля тэгов головного узла можно оставить неопределенными.

1. Прошивка правых связей. Вводится дополнительный глобальный указатель *y*(указатель на узел, предшествующий текущему узлу). Указатель на текущий узел *p* устанавливаем на корень дерева (*p := HEAD^.llink* ).

Выполняется симметричный обход дерева. При обработке каждого узла проверяется: если *p^.right <>nil*, то *p^.rtag:=true;* иначе *p^.rtag:=false; y:=p;* (указатель на предшественника). В любом случае выполняем.

Программный код процедур симметричного обхода *sim\_print* и прошивки правых связей *rightsew* будет выглядеть так:

*procedure sim\_print(var x:pt);*

*procedure rightsew( var p:pt);*

*begin*

*if y <> nil then*

*begin*

*if y^.right=nil then*

*begin*

*y^.rtag := false;*

*y^.right := p;*

*end*

*else y^.rtag := true;*

*end;*

*y:=p;*

*end;*

*begin*

*if x<> nil then*

*begin*

*sim\_print(x^.left);*

*rightsew(x);*

*sim\_print(x^.right);*

*end;*

*end;*

**39. Структура узла прошитого дерева. Процедура обхода прямо прошитого бинарного дерева.**

Рассмотрим алгоритм обхода прошитого бинарного дерева. Пусть HEAD – указатель на головной узел прошитого дерева,*p*– указатель на текущий узел. Тогда алгоритм симметричного обхода прошитого дерева можно сформулировать следующим образом:

1. Переход к корню дерева ( *p := HEAD^.left*).
2. До тех пор, пока *p^.left<>nil*, повторять:*p := p^.left*,то есть идти по левой ветви до самого левого узла.
3. Обработка узла *p*, например, печать*p^.info*.
4. Если *p^.rtag* равен*false*, то*p := p^.right* и переход к шагу 3 ( к преемнику). Иначе*p:= p^.right* и переход к шагу 2.

+

Алгоритм заканчивает работу, когда *p* станет равным HEAD.

**40. Метод представления сообщений кодами Хаффмана.**

Идея алгоритма состоит в следующем: зная вероятности появления символов в сообщении, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью ставятся в соответствие более короткие коды.

**41. Этапы создания дерева Хаффмана для заданных сообщений.**

Классический алгоритм Хоффмана на входе получает таблицу частот встречаемости символов в сообщении. Далее на основании этой таблицы строится дерево кодирования Хоффмана.

1) Символы входного алфавита образуют список свободных узлов. Каждый лист имеет вес, который может быть равен либо вероятности, либо количеству вхождений символа в сжимаемое сообщение. 2) Выбираются два свободных узла дерева с наименьшими весами. 3) Создается их родитель с весом, равным их суммарному весу. 4) Родитель добавляется в список свободных узлов, а два его потомка удаляются из этого списка. 5) Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой бит 0. Битовые значения ветвей, исходящих от корня, не зависят от весов потомков. 6) Шаги, начиная со второго, повторяются до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

Допустим, у нас есть следующая таблица частот:

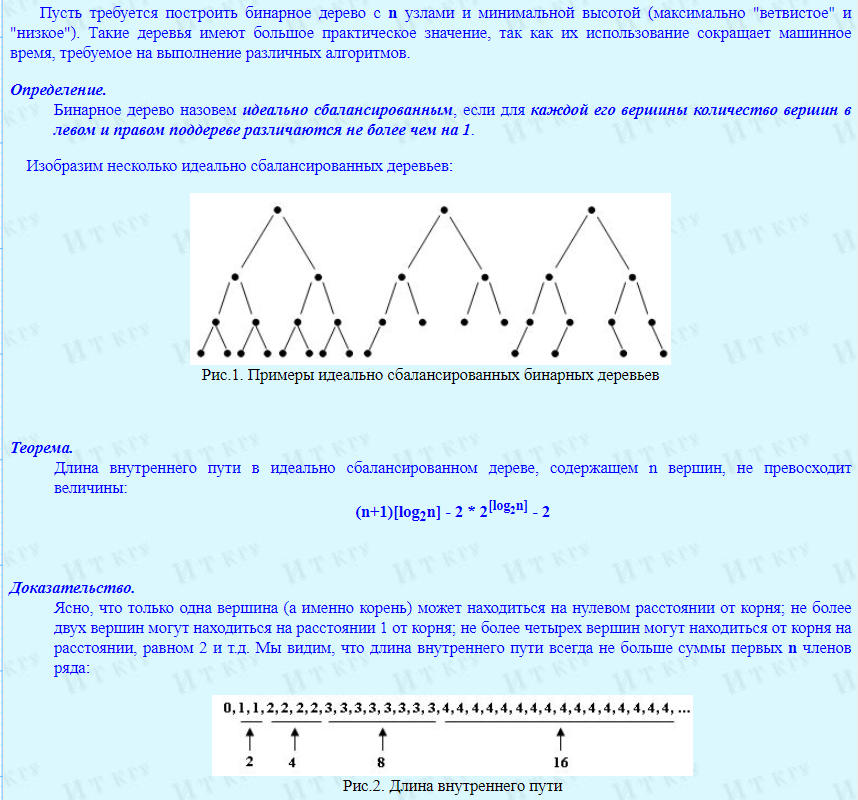
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Символ | А | Б | В | Г | Д |
| Частота | 15 | 7 | 6 | 6 | 5 |

Этот процесс можно представить как построение [дерева](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_(%D1%82%D0%B5%D0%BE%D1%80%D0%B8%D1%8F_%D0%B3%D1%80%D0%B0%D1%84%D0%BE%D0%B2)), корень которого - символ с суммой вероятностей объединенных символов, получившийся при объединении символов из последнего шага, его n0 потомков - символы из предыдущего шага и т.д. Чтобы определить код для каждого из символов, входящих в сообщение, мы должны пройти путь от листа дерева, соответствующего текущему символу, до его корня, накапливая биты при перемещении по ветвям дерева (первая ветвь в пути соответствует младшему биту). Полученная таким образом последовательность битов является кодом данного символа, записанным в обратном порядке. Для данной таблицы символов коды Хаффмана будут выглядеть следующим образом.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Символ | А | Б | В | Г | Д |
| Код | 0 | 100 | 101 | 110 | 111 |

Поскольку ни один из полученных кодов не является префиксом другого, они могут быть однозначно декодированы при чтении их из потока. Кроме того, наиболее частый символ сообщения А закодирован наименьшим количеством бит, а наиболее редкий символ Д - наибольшим.

**42. Идеально сбалансированное бинарное дерево. Правила построения. Достоинства и недостатки. Привести пример такого дерева.**

****

**43.Сбалансированное бинарное дерево. Сравнить с идеально сбалансированным. Привести пример таких деревьев.**

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков (оба потомка которых равны NULL) называются листьями.

Сбалансированное бинарное дерево поиска — это бинарное дерево поиска с логарифмической высотой. Данное определение скорее идейное, чем строгое. В сбалансированном бинарном дереве поиска операции поиска, вставки и удаления выполняются за логарифмическое время (так как путь к любому листу от корня не более логарифма). В вырожденном случае несбалансированного бинарного дерева поиска, например, когда в пустое дерево вставлялась отсортированная последовательность, дерево превратится в линейный список, и операции поиска, вставки и удаления будут выполняться за линейное время. Поэтому балансировка дерева крайне важна. Технически балансировка осуществляется поворотами частей дерева при вставке нового элемента, если вставка данного элемента нарушила условие сбалансированности.

Сбалансированное бинарное дерево поиска применяется, когда необходимо осуществлять быстрый поиск элементов, чередующийся со вставками новых элементов и удалениями существующих. В случае, если набор элементов, хранящийся в структуре данных фиксирован и нет новых вставок и удалений, то массив предпочтительнее. Потому что поиск можно осуществлять алгоритмом бинарного поиска за то же логарифмическое время, но отсутствуют дополнительные издержки по хранению и использованию указателей. Например, в С++ ассоциативные контейнеры set и map представляют собой сбалансированное бинарное дерево поиска.

**44. Вставка элемента в АВЛ-дерево. Привести пример.**

АВЛ-дерево - сбалансированное по высоте [двоичное дерево поиска](https://ru.wikipedia.org/wiki/%D0%94%D0%B2%D0%BE%D0%B8%D1%87%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0): для каждой его вершины высота её двух поддеревьев различается не более чем на 1. Максимальная высота АВЛ-дерева при заданном числе узлов: h≤[1.45 log2(n+2)].

Относительно АВЛ-дерева балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины. Операцию балансировки требуется выполнять всякий раз, когда в дереве происходят изменения. В операции вставки перед непосредственным возвратом из процедуры необходимо проверять, является ли дерево сбалансированным, и если нет, то осуществлять балансировку.

Алгоритм добавления вершины:

Показатель сбалансированности в дальнейшем будем интерпретировать как разность между высотой левого и правого поддерева, а алгоритм будет основываться на типе TAVLTree, описанном выше. Непосредственно при вставке (листу) присваивается нулевой баланс. Процесс включения вершины состоит из трех частей: 1) Прохода по пути поиска, пока не убедимся, что ключа в дереве нет. 2) Включения новой вершины в дерево и определения результирующих показателей балансировки. 3) «Отступления» назад по пути поиска и проверки в каждой вершине показателя сбалансированности. Если необходимо — балансировка.

Будем возвращать в качестве результата функции, уменьшилась высота дерева или нет. Предположим, что процесс из левой ветви возвращается к родителю (рекурсия идет назад), тогда возможны три случая: { hl — высота левого поддерева, hr — высота правого поддерева } Включение вершины в левое поддерево приведет к:

1) hl < hr: выравняется hl = hr. Ничего делать не нужно.

2) hl = hr: теперь левое поддерево будет больше на единицу, но балансировка пока не требуется.

3) hl > hr: теперь hl — hr = 2, — требуется балансировка.

В третьей ситуации требуется определить балансировку левого поддерева. Если левое поддерево этой вершины (Tree^.left^.left) выше правого (Tree^.left^.right), то требуется большое правое вращение, иначе хватит малого правого. Аналогичные (симметричные) рассуждения можно привести и для включение в правое поддерево.

Нерекурсивный алгоритм сложнее рекурсивного: 1) Находится место вставки и вершина, высота которой не изменится при вставке (это вершина, у которой высота левого поддерева не равна высоте правого; будем называть её PrimeNode) 2) Выполняется спуск от PrimeNode до места вставки с изменением балансов 3) Выполняется ребалансировка PrimeNode при наличии переполнения.

**45. Основные определения ориентированных графов: вершина, дуга, путь, цикл. Помеченный орграф. Привести примеры. Матрицы смежности и инцидентности.**

Дуга – это направленное ребро в орграфе.

Путь в графе - это последовательность вершин (без повторений), в которой любые две соседние вершины смежны.

Цикл - это замкнутый путь.

Во многих приложениях удобно к вершинам и дугам орграфа присоединить какую-либо информацию. Для этих целей используется помеченный орграф, т. е. орграф, у которого каждая дуга и/или каждая вершина имеет соответствующие метки. Меткой может быть имя, вес или стоимость (дуги), или значение данных какого-либо заданного типа.

На рис. 5.2 показан помеченный орграф, в котором каждая дуга имеет буквенную метку. Этот орграф имеет интересное свойство: любой цикл, начинающийся в вершине 1, порождает последовательность букв а и b, в которой всегда четное количество букв а и b.

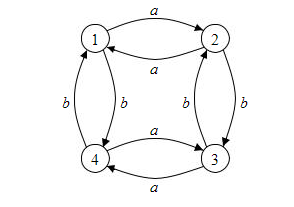


Рис. 5.2. Помеченный орграф

Инцидентность - это когда вершина a является либо началом либо концом ребра e. Две вершины называются инцидентными, если у них есть общее ребро.

Для того, чтобы задать граф аналитически, множества V вершин графа и множества U рёбер графа, которые фигурировали в [определении графа](https://function-x.ru/graphs1_relations.html), будет недостаточно. Потребуется ещё и множество P троек вида (a, u, b), указывающих какую пару a, b элементов множества вершин V соединяет тот или иной элемент u множества рёбер U графа. Элементы множества P называются инциденциями графа. Вот мы и подошли к одному из первых понятий теории графов - инцидентности.

Смежность вершин графа - это когда две вершины графа соединены ребром.

Зададимся вопросом: можно ли поместить слона в компьютер? Ответ: можно, если слона смоделировать в виде графа, в котором вершинами являются части его тела, а рёбра соединяют те части тела, которые соединены в слоне как биологическом объекте. При этом получившийся граф должен быть представлен в памяти компьютера в понятном компьютеру виде.

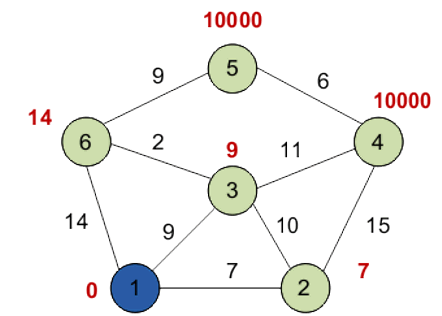
В связи с широким применением [графов](https://function-x.ru/graphs1_relations.html) в программировании и информационных технологиях вообще возникает вопрос о представлении графа в виде структуры данных. Различные способы представления графов в памяти компьютера отличаются объёмом занимаемой памяти и скоростью выполнения операций над графами.

**46. Нахождение кратчайшего пути на орграфе с помощью алгоритма Дейкстры. Привести пример.**

Первый шаг

Минимальную метку имеет вершина 1. Её соседями являются вершины 2, 3 и 6. Обходим соседей вершины по очереди.

Первый сосед вершины 1 – вершина 2, потому что длина пути до неё минимальна. Длина пути в неё через вершину 1 равна сумме кратчайшего расстояния до вершины 1 (значению её метки) и длины ребра, идущего из 1-й во 2-ю, то есть 0 + 7 = 7. Это меньше текущей метки вершины 2 (10000), поэтому новая метка 2-й вершины равна 7.



Аналогично находим длины пути для всех других соседей (вершины 3 и 6).

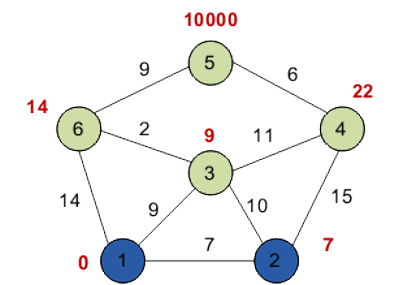
Все соседи вершины 1 проверены. Текущее минимальное расстояние до вершины 1 считается окончательным и пересмотру не подлежит. Вершина 1 отмечается как посещенная.

Второй шаг

Шаг 1 алгоритма повторяется. Снова находим «ближайшую» из непосещенных вершин. Это вершина 2 с меткой 7.

Снова пытаемся уменьшить метки соседей выбранной вершины, пытаясь пройти в них через 2-ю вершину. Соседями вершины 2 являются вершины 1, 3 и 4.

Вершина 1 уже посещена. Следующий сосед вершины 2 — вершина 3, так как имеет минимальную метку из вершин, отмеченных как не посещённые. Если идти в неё через 2, то длина такого пути будет равна 17 (7 + 10 = 17). Но текущая метка третьей вершины равна 9, а 9 < 17, поэтому метка не меняется.

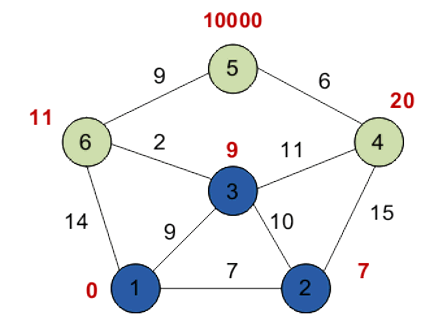


Ещё один сосед вершины 2 — вершина 4. Если идти в неё через 2-ю, то длина такого пути будет равна 22 (7 + 15 = 22). Поскольку 22<10000, устанавливаем метку вершины 4 равной 22.

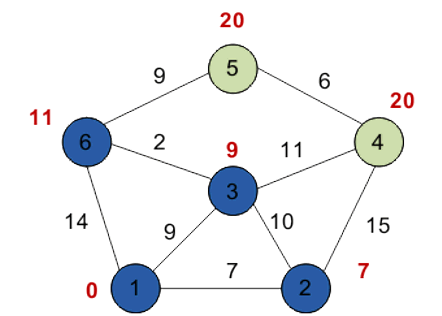
Все соседи вершины 2 просмотрены, помечаем её как посещенную.

Третий шаг

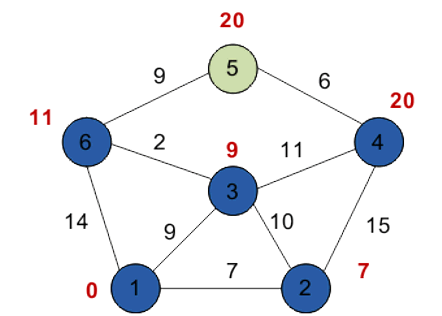
Повторяем шаг алгоритма, выбрав вершину 3. После её «обработки» получим следующие результаты.



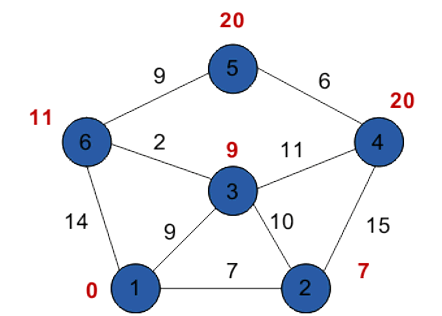
Четвертый шаг



Пятый шаг



Шестой шаг



Таким образом, кратчайшим путем из вершины 1 в вершину 5 будет путь через вершины 1 — 3 — 6 — 5, поскольку таким путем мы набираем минимальный вес, равный 20.

Займемся выводом кратчайшего пути. Мы знаем длину пути для каждой вершины, и теперь будем рассматривать вершины с конца. Рассматриваем конечную вершину (в данном случае — вершина 5), и для всех вершин, с которой она связана, находим длину пути, вычитая вес соответствующего ребра из длины пути конечной вершины.

Так, вершина 5 имеет длину пути 20. Она связана с вершинами 6 и 4.

Для вершины 6 получим вес 20 — 9 = 11 (совпал).

Для вершины 4 получим вес 20 — 6 = 14 (не совпал).

Если в результате мы получим значение, которое совпадает с длиной пути рассматриваемой вершины (в данном случае — вершина 6), то именно из нее был осуществлен переход в конечную вершину. Отмечаем эту вершину на искомом пути.

Далее определяем ребро, через которое мы попали в вершину 6. И так пока не дойдем до начала.

Если в результате такого обхода у нас на каком-то шаге совпадут значения для нескольких вершин, то можно взять любую из них — несколько путей будут иметь одинаковую длину.

**47. Нахождение кратчайших путей на орграфе между парами вершин с помощью алгоритма Флойда. Привести пример.**

**48. Транзитивное замыкание на орграфе. Привести пример.**

**49. Нахождение центра орграфа. Привести пример.**

*Центром орграфа* G называется вершина с минимальным эксцентриситетом. Дру­гими словами, центром орграфа является вершина, для кот 828b12fi орой максимальное рас­стояние (длина пути) до других вершин минимально.

Рассмотрим помеченный орграф, показанный на рис. 11. В этом графе вершины имеют следующие эксцентриситеты.

|  |  |
| --- | --- |
| Вершина | Эксцентриситет |
| A | ∞ |
| B | 6 |
| C | 8 |
| D | 5 |
| E | 7 |

Из этой таблицы видно, что центром данного орграфа является вершина D.

Найти центр орграфа сравнительно просто. Пусть С матрица стоимостей для

орграфа G.

1. Сначала применим функцию FLOYD() к матрице С для вычисления матрицы А, содержащей все кратчайшие пути орграфа G.

2. Находим максимальное значение в каждом столбце i матрицы А. Это значение равно эксцентриситету вершины i.

3. Находим вершину с минимальным эксцентриситетом. Она и будет центром графа *G*.

|  |  |
| --- | --- |
|  |  |
|  |  |

Находим вершину с минимальным эксцентриситетом. Она и будет центром графа G.

Время выполнения этого процесса определяется первым шагом, для которого время имеет порядок O(n3). Второй шаг требует времени порядка O(n2), а тре­тий — O(n).

**void CENTER (ORGRAF \* GR,int r)**

**}**

**// Алгоритм Флойда**

**for (int k=0;k<r;k++)**

**for (i=0;i<r;i++)**

**for (int j=0;j<r;j++)**

**if ((C[i][k]+C[k][j])<C[i][j])**

**printf ("\n\n Матрица кратчайших путей \n");**

**for (i=0;i<r;i++)**

**// Нахождение эксцентристет для каждой вершины графа**

**float \*EXCENTR=(float \*)malloc(r\*sizeof(float));**

**for (int j=0;j<r;j++)**

**// Нахождение центра ориентированного графа**

**float min=EXCENTR[0];**

**int c\_og=0;**

**for (i=1;i<r;i++)**

**if (EXCENTR[i]<min)**

**printf ("\n Центром ориентированного графа является вершина %i . Эксцентристет - %5.2f",c\_og,min);**

**free (EXCENTR);**

**free(C);**

**}**

**50. Особенности алгоритмов для внешней памяти. Хешированные файлы.**

**51.Особенности алгоритмов для внешней памяти. Индексированные файлы.**

**52. Особенности алгоритмов для внешней памяти. Построение В-дерева.**

**53. Поиск заданного элемента в В-дереве.**

**54. Вставка заданного элемента в В-дерево.**

**55. Удаление заданного элемента из В-дерева.**