**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:

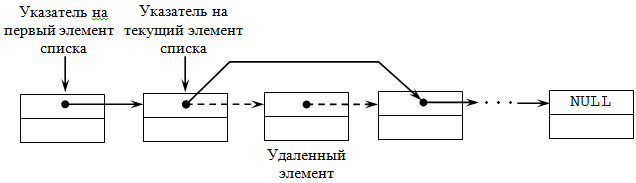
* создание списка;
* печать (просмотр) списка;
* вставка элемента в список;
* удаление элемента из списка;
* поиск элемента в списке
* проверка пустоты списка;
* удаление списка.

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](https://intuit.ru/studies/courses/648/504/lecture/11456?page=2#image.29.2)).

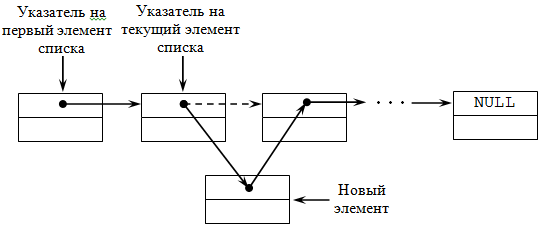


Рис. 29.2. Вставка элемента в однонаправленный список

/\*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;

}

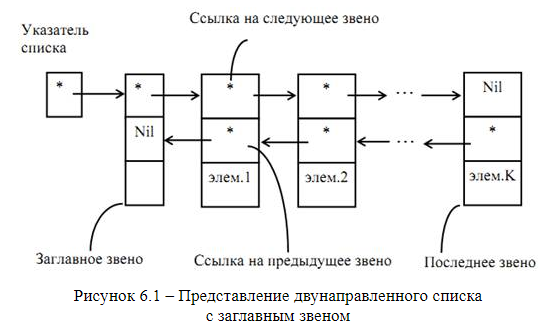
}

return Head;

}

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

Пусть в программе имеется описание типа, приведенное в примере 6.1.



*Алгоритм вставки элемента в указанное место двунаправленного кольцевого списка:*

1) Порождение нового звена.

2) Занесение вставляемого элемента в информационное поле порожденного звена.

3)  Занесение в поле *Adrcled* порожденного звена ссылки на следующий элемент из звена, предшествующего вставляемому.

4) Занесение в поле *Adrpred* порожденного звена ссылки на предыдущий элемент из звена, следующего за вставляемым.

5) Занесение в поле *Adrpred* следующего за вставляемым звена ссылки на вставляемое звено.

6) Занесение в поле *Adrcled* предшествующего звена ссылки на вставляемое звено.

Действия, необходимые для вставки элемента в двунаправленный кольцевой список с заглавным звеном, схематически поясняют рисунки 6.4 –6.5.

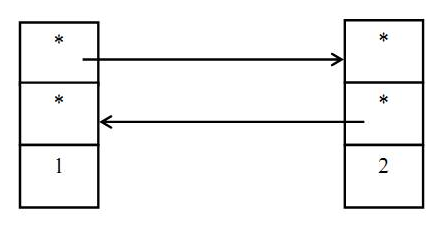


Рисунок 6.4 – Фрагмент исходного двунаправленного списка

Пусть в исходный список после элемента 1 вставляется элемент 3. Соответствующий фрагмент результирующего списка будет иметь вид, который изображает рисунок 6.5.

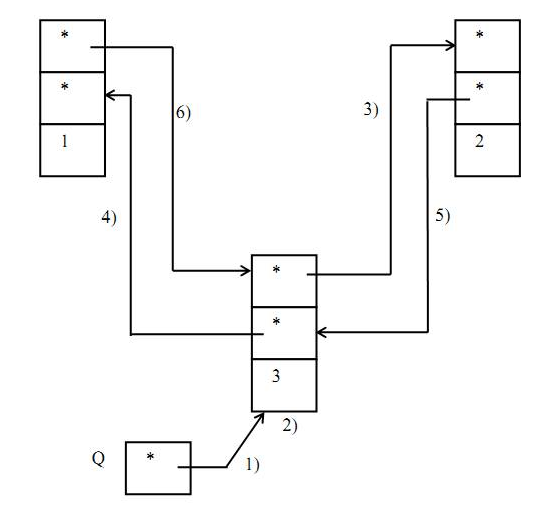


Рисунок 6.5 – Фрагмент результирующего двунаправленного списка

На данном рисунке номера 1) – 6) соответствуют номерам этапов алгоритма вставки.

*Пример 6.2.*Процедура вставки.

*Procedure Vstav (Elem: <Тип\_элемента\_списка>; Predzv: Adr2);*

                                                      {Elem – вставляемый элемент, Predzv –

                                                      ссылка на предшествующий элемент}

*Var Q: Adr2;*

*Begin*

{1}*New (Q);*

{2}*Q^.Element := Elem;*

{3}*Q^.Adrcled := Predzv^.Adrcled;*

{4}*Q^.Adrpred := Predzv;*

{5}*Predzv^.Adrcled^.Adrpred := Q;*

{6}*Predzv^.Adrcled := Q;*

*End;*

В данном примере номера операторов, приведенные в комментариях, соответствуют номерам этапов алгоритма вставки. Реализация этапа 4 в примере отличается от этапа 4 алгоритма вставки, поскольку ссылка на предшествующее звено в процедуру *Vstav*передается в качестве параметра.

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

Пусть в программе имеется описание типа, приведенное в примере 6.1.

*Алгоритм удаления элемента:*

1) Занесение в поле *Adrpred* следующего за удаляемым звена ссылки на предшествующее удаляемому звено из поля *Adrpred*удаляемого звена;

2) Занесение в поле *Adrcled* предшествующего удаляемому звена ссылки на следующее за удаляемым звено из поля *Adrcled*удаляемого звена;

3) Уничтожение удаляемого звена.

На двух следующих рисунках приведены схематические пояснения к удалению элемента из двунаправленного списка.

Пусть в изображенном фрагменте исходного списка удаляется звено 2 (рисунок 6.7).

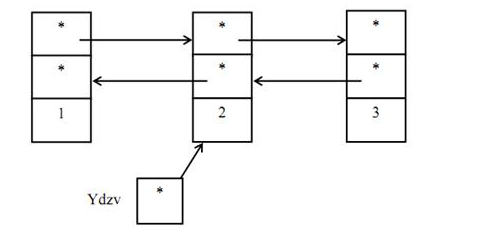


Рисунок 6.7 – Фрагмент исходного двунаправленного списка

На данном рисунке через *Ydzv* обозначена ссылка на удаляемое звено.

Результат удаления звена 2 выглядит в соответствии с рисунком 6.8. Номера связей на нем соответствуют номерам этапов алгоритма удаления.

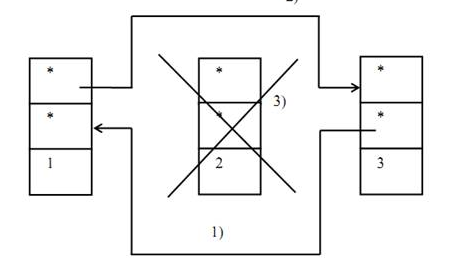


Рисунок 6.8 – Результат удаления звена 2

*Пример 6.4.*Процедура удаления звена из двунаправленного списка. Номера операторов соответствуют номерам этапов алгоритма удаления звена.

*Procedure Udalen (Ydzv: Adr2);*           {Ydzv – ссылка на удаляемое звено}

*Begin*

{1}*Ydzv^.Adrcled^.Adrpred := Ydzv^.Adrpred;*

{2}*Ydzv^.Adrpred^.Adrcled := Ydzv^.Adrcled;*

{3}*Dispose (Udzv);*

*End;*

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

Поиск элемента в двунаправленном кольцевом списке аналогичен поиску элемента в динамической цепочке.

Особенность поиска заключается в том, что в кольцевом списке формально нет последнего элемента, так как каждый элемент имеет ссылку на следующий. Это нужно учитывать при организации цикла поиска.

*Пример 6.5.* Логическая функция поиска элемента в двунаправленном кольцевом списке с заглавным звеном. Если искомый элемент в списке есть, возвращаемое значение функции равно *True*, а параметру *Iskadr* присваивается ссылка на звено, содержащее данный элемент.

*Function Poisk (Adr: Adr2; Elem: <Тип\_элемента\_списка>; Var Iskadr: Adr2)****:***

*Boolean;*{Adr – ссылка на заглавное звено;

                                                         Elem – искомый элемент;

                                                         Iskadr – адрес искомого элемента}

*Var*

*P, Q: Adr2;*{В Р будет храниться адрес

                                                          заглавного звена,

                                                          в Q – адрес текущего звена}

*B: Boolean;*

*Begin*

*B := False;*{B – логическая переменная

                                                         (факт наличия искомого элемента)}

*P := Adr;*{В P занесен адрес заглавного звена}

*Iskadr := Nil;*

*Q:=P^.Adrcled;*{В Q занесен адрес первого звена}

*While (P<>Q) And Not B Do*{Поиск, пока не дошли

                                                                до заглавного звена и

                                                                не нашли искомый элемент}

*Begin*

*If Q^.Element = Elem Then*{Найден искомый элемент}

*Begin*

*B := True;*

*Iskadr := Q*{Адрес искомого элемента}

*End;*

*Q := Q^.Adrcled*{Переход к следующему звену}

*End;*

*Poisk := B*{Возвращаемое значение}

*End;*

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

Основная идея метода заключается в том, что множествово данных разбивается на конечное число классов. Для В классов от 0 до В-1 строится хеш-функция Н, такая что для любого элемента Х из исходного множества функция H(x) принимает целочисленное значение от 0..В-1, соответствующее классу, которому принадлежит элемент Х.

Х - ключ хэш функции. H(x) – хеш-значением Х, а классы – сегментами. Таблица сегментов (массив), проиндексированный номерами сегментов 0, 1, …, В – 1 содержит заголовки для В списков. Элемент Х i – ого списка – это элемент исходного множества, для которого H(x)=i.

Идеальной хеш-функцией является такая, которая для любых двух неодинаковых ключей выдает неодинаковые адреса. Если сегменты одинаковы, то списки должны быть max короткими. Средняя длина – N/B. N - количество элементов в исходном множестве. **Н** следует выбирать, чтобы поровну распределяла элементы по сегментам. **Пример**: Представление символов в виде целых чисел. В качестве Н используется **Ord**. Если Х – это ключ длиной j символов (тип данных ключей определен как строка символов), тогда можно использовать хеш-функцию, в которой суммируются целочисленные коды всех символов ключа результат делится на В и берётся остаток от деления, который будет целым числом из интервала от 0 до В - 1.

**int hash(char \*v, int M)**

**{ int h = 0, a = 127;**

**for (; \*v != 0; v++)**

**h = (a\*h + \*v) % M;**

**return h;**

**}**

begin

sum:=0;

for i:=1 to 10 do

**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.**

Коллизии осложняют использование хеш-таблиц, так как нарушают однозначность соответствия между хеш-кодами и данными. Способы предотвращения сложностей: метод цепочек для открытого хеширования, метод открытой адресации для закрытого хеширования. **Метод цепочек.** Технология сцепления элементов состоит в том, что элементы множества, которым соответствует одно и то же хеш-значение, связываются в цепочку-список. В позиции номер i хранится указатель на голову списка тех элементов, у которых хеш-значение ключа равно i; если таких элементов в множестве нет, в позиции i записан NULL. Каждая ячейка массива является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента. Операции поиска или удаления данных требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом. Для добавления данных нужно добавить элемент в конец или начало соответствующего списка, и, в случае если коэффициент заполнения станет слишком велик, увеличить размер массива и перестроить таблицу. При предположении, что каждый элемент может попасть в любую позицию таблицы с равной вероятностью и независимо от того, куда попал любой другой элемент, среднее время работы операции поиска элемента составляет O(1+k), где k – коэффициент заполнения таблицы. **Метод открытой адресации.** В отличие от хеширования с цепочками, при открытой адресации никаких списков нет, а все записи хранятся в самой хеш-таблице. Каждая ячейка таблицы содержит либо элемент динамического множества, либо NULL. В этом случае, если ячейка с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку до тех пор, пока не будет найден ключ K или пустая позиция в таблице. Для вычисления шага можно также применить формулу, которая и определит способ изменения шага.

При любом методе разрешения коллизий необходимо ограничить длину поиска элемента. Если для поиска элемента необходимо более 3–4 сравнений, то эффективность использования такой хеш-таблицы пропадает и ее следует найти другую хеш-функцию, чтобы минимизировать количество сравнений для поиска элемента. Для успешной работы алгоритмов поиска, последовательность проб должна быть такой, чтобы все ячейки хеш-таблицы оказались просмотренными ровно по одному разу.

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

**Хеш-таблицей** называется структура данных, предназначенная для реализации ассоциативного массива, такого в котором адресация реализуется посредством хеш-функции. **Хеш-функция** – это функция, преобразующая ключ **key** в некоторый индекс **i**равный**h(key)**, где h(key) – **хеш-код** (хеш-сумма, хеш) key. Весь процесс получения индексов хеш-таблицы называется **хешированием**.

**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:

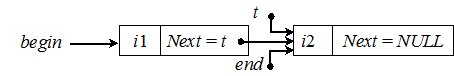
end -> Next = t;

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.

Сцщность 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.**

В инфиксной записи операция разделяет два операнда, в постфиксной — операция следует за двумя операндами, в префиксной — операция предшествует двум операндам. Примеры записи арифметических выражений в различных формах приведены в табл. 1.1.

*Таблица 1.1*

Формы записи арифметических выражений

|  |  |  |
| --- | --- | --- |
| Инфиксная запись | Постфиксная запись | Префиксная запись |
| А+В | АВ+ | +АВ |
| А+В-С | АВ+С- | +А-ВС |
| А\*В+С | АВ\*С+ | +\*АВС |
| А+В\*С | АВС\*+ | +А\*ВС |
| (А+В)\*С | АВ+С\* | \*+АВС |

*Перевод инфиксной формы записи выражения в постфиксную (польскую).* Инфиксное выражение сканируется слева направо, и в зависимости от вида очередного элемента выражения выполняется одно из действий, представленных в табл. 1.2.

*Таблица 1.2*

Перевод инфиксной формы записи выражения в постфиксную (польскую)

|  |  |
| --- | --- |
| Элемент выражения | Действие |
| Открывающая скобка | Вталкивание открывающей скобки в стек. |
| Операнд | Запись операнда в постфиксную строку. |
| Закрывающая скобка | Выталкивание элементов из стека до первой открывающей скобки и запись их в постфиксную строку, затем выталкивание самой открывающей скобки без записи ее в постфиксную строку. Если перед выполнением этой операции стек оказался пустым, значит, для данной закрывающей скобки не было парной открывающей, т.е. возникла исключительная ситуация. |

|  |  |
| --- | --- |
| Элемент выражения | Действие |
| Операция | Если стек не пуст и приоритет операции ниже либо такой же (<), как у верхней операции в стеке (открывающая скобка в данном контексте считается операцией с *низшим* приоритетом), то выталкивание элементов из стека до операции с меньшим приоритетом (реально: для операций «+» и «—» до «(», а для операций «\*» и «/» до «+»,  «—» или «(») или до опустошения стека и запись их в постфиксную строку; в противном случае стек не изменяется. Затем вталкивание операции в стек. |

После просмотра выражения выталкиваются из стека и записываются в постфиксную строку оставшиеся операции.

*Вычисление значения выражения по его постфиксной записи.* Постфиксное выражение сканируется слева направо и обрабатывается в соответствии с алгоритмом, приведенным в табл. 1.3.

*Таблица 1.3*

Вычисление значения выражения по его постфиксной записи

|  |  |
| --- | --- |
| Элемент  выражения | Действие |
| Операнд | Вталкивание операнда в стек |
| Операция | Выталкивание из стека двух элементов, выполнение операции с ними, запись результата в стек. |

Результат выполнения последней операции является результатом вычисления всего выражения.

*Перевод из инфиксной формы записи выражения в префиксную.* Инфиксное выражение сканируется справа налево, и префиксная строка строится также справа налево. Алгоритм преобразования такой же, как и при преобразовании в постфиксную форму, только открывающие скобки меняются на закрывающие и, наоборот. При определении приоритета операции отношение «<» изменяется на «<», чтобы равноприоритетные операции выполнялись слева направо.

*Вычисление значения выражения по его префиксной записи.* Алгоритм вычисления полностью совпадает с алгоритмом вычисления по постфиксной записи, но префиксная строка сканируется справа налево.

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

**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.

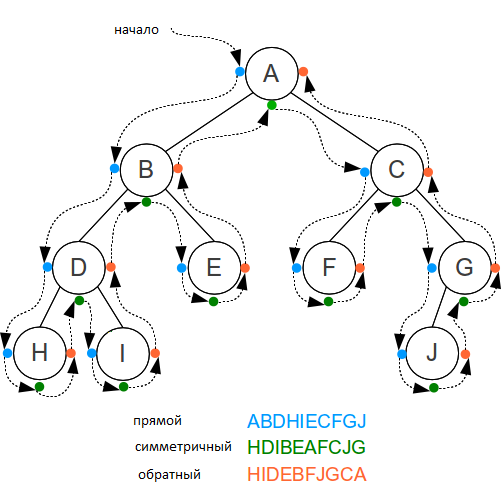
**28. Описание вершины дерева. Понятие бинарного дерева поиска.** Процедура вставки элемента в бинарное дерево поиска.

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** — вид **обхода** графа, обусловливающий процесс посещения каждого узла структуры **дерева** данных ровно один раз.

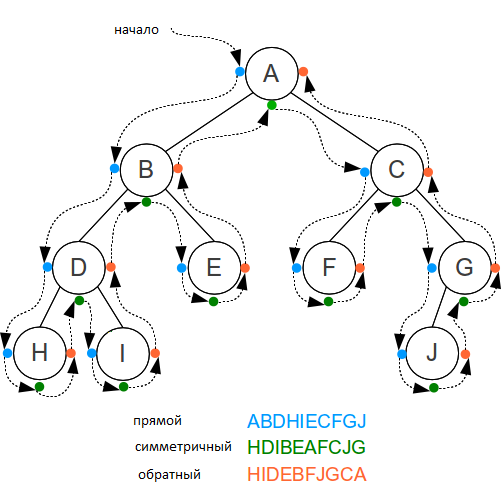
Прямой обход идет в следующем порядке: корень, левый потомок, правый потомок. В коде рекурсивной функции соответствующего обхода сохраняется соответствующий порядок вызовов (порядок строк кода), где вместо корня идет вызов данной рекурсивной функции.  
  
Если нам дано изображение дерева, и нужно найти его обходы, то может помочь следующая техника (см. рис. 5). Обводим дерево огибающей замкнутой кривой (начинаем идти слева вниз и замыкаем справа вверх). Прямому обходу будет соответствовать порядок, в котором огибающая, двигаясь от корня впервые проходит рядом с узлами слева. В коде рекурсивного вызова прямого обхода идет: вызов, левый, правый.



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

**Обход** **дерева** — вид **обхода** графа, обусловливающий процесс посещения каждого узла структуры **дерева** данных ровно один раз.

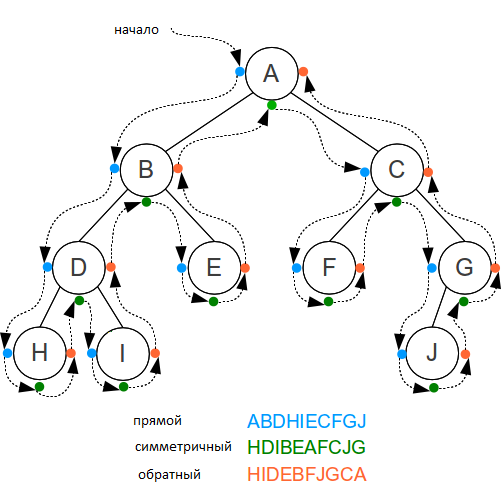
Прямой обход идет в следующем порядке: корень, левый потомок, правый потомок. Симметричный — левый потомок, корень, правый потомок. В коде рекурсивной функции соответствующего обхода сохраняется соответствующий порядок вызовов, где вместо корня идет вызов данной рекурсивной функции.

Если нам дано изображение дерева, и нужно найти его обходы, то может помочь следующая техника (см. рис. 5). Обводим дерево огибающей замкнутой кривой (начинаем идти слева вниз и замыкаем справа вверх). Для симметричного обхода порядок, в котором огибающая, двигаясь от корня впервые проходит рядом с узлами снизу. В коде рекурсивного вызова прямого обхода идет: вызов, левый, правый. Симметричного – левый,вызов,правый.   
  
  
Для бинарных деревьев поиска симметричный обход проходит все узлы в отсортированном порядке. Если мы хотим посетить узлы в обратно отсортированном порядке, то в коде рекурсивной функции симметричного обхода следует поменять местами правого и левого потомка.

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

**Обход** **дерева** — вид **обхода** графа, обусловливающий процесс посещения каждого узла структуры **дерева** данных ровно один раз.

Обратный – левый потомок, правый потомок, корень. В коде рекурсивной функции соответствующего обхода сохраняется соответствующий порядок вызовов (порядок строк кода), где вместо корня идет вызов данной рекурсивной функции.

Если нам дано изображение дерева, и нужно найти его обходы, то может помочь следующая техника (см. рис. 5). Обводим дерево огибающей замкнутой кривой (начинаем идти слева вниз и замыкаем справа вверх). Для обратного обхода порядок, в котором огибающая, двигаясь от корня впервые проходит рядом с узлами справа.   


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

В древовидной структуре число потомков вершины называется ее степенью. Максимальное значение этих степеней называется степенью дерева. Наибольшую популярность в программировании и вычислительной технике получили *бинарные (двоичные) деревья*, у которых степень дерева равна двум*.* В этом случае вершина дерева может иметь не более двух потомков, называемых *левым* и *правым* сыновьями. В отдельный подкласс *бинарных деревьев*выделены*деревья* *поиска*. Они характеризуются тем, что значение информационного поля, связанного с вершиной дерева, больше любого соответствующего значения из левого поддерева и меньше, чем содержимое любого узла его правого поддерева.

Схематично двоичное дерево можно представить как набор вершин, соединенных стрелками (ветвями, рисунок 11.4). Из каждой вершины выходит не более двух ветвей, направленных влево-вниз или вправо-вниз. В каждую вершину, помимо одной, входит одна стрелка. Вершина, в которую не входит ни одна стрелка, называется *корнем дерева*. Вершины, из которых не выходит ни одна стрелка, называются *листьями*.

Есть три операции обхода узлов дерева, отличающиеся порядком обхода узлов:

* inorderTraversalinorderTraversal — обход узлов в отсортированном порядке,
* preorderTraversalpreorderTraversal — обход узлов в порядке: вершина, левое поддерево, правое поддерево,
* postorderTraversalpostorderTraversal — обход узлов в порядке: левое поддерево, правое поддерево, вершина.

 Для поиска элемента в бинарном дереве поиска можно воспользоваться следующей функцией, которая принимает в качестве параметров корень дерева и искомый ключ. Для каждого узла функция сравнивает значение его ключа с искомым ключом. Если ключи одинаковы, то функция возвращает текущий узел, в противном случае функция вызывается рекурсивно для левого или правого поддерева. Узлы, которые посещает функция образуют нисходящий путь от корня, так что время ее работы O(h)O(h), где hh — высота дерева.

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

Для включения записи в дерево нужно найти в нем вершину, к которой можно присоединить новую вершину, соответствующую включаемой записи. Алгоритм поиска нужной вершины аналогичен алгоритму поиска вершины с заданным ключом (см. пример 12.1). Нужная вершина найдена, если в качестве очередной ссылки, определяющей ветвь продолжения поиска, окажется ссылка *Nil*. В качестве заданного ключа в данном случае используется ключ включаемой вершины.

Таким образом, для поиска вершины, к которой можно присоединить включаемую запись, можно воспользоваться алгоритмом поиска вершины с заданным ключом, реализованным в примере 12.1. Такая вершина найдена, если *В = False*. В этом случае в формальном параметре *Rez* процедуры *Poisk* находится адрес вершины, к которой можно подсоединить включаемую вершину.

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

Для удаления узла из бинарного дерева поиска нужно рассмотреть три возможные ситуации. Если у узла нет дочерних узлов, то у его родителя нужно просто заменить указатель на nullnull. Если у узла есть только один дочерний узел, то нужно создать новую связь между родителем удаляемого узла и его дочерним узлом. Наконец, если у узла два дочерних узла, то нужно найти следующий за ним элемент (у этого элемента не будет левого потомка), его правого потомка подвесить на место найденного элемента, а удаляемый узел заменить найденным узлом. Таким образом, свойство бинарного дерева поиска не будет нарушено. Данная реализация удаления не увеличивает высоту дерева. Время работы алгоритма — O(h)O(h).

 а) *Самый правый элемент левого от удаляемой вершины поддерева*.

б) *самый* *левый элемент правого от удаляемой вершины поддерева.*

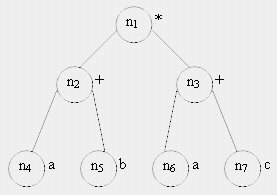
**34. Помеченные деревья. Правила соответствия меток деревьев элементам выражений. Привести пример прямого, обратного и симметричного обходов помеченного дерева.**

Часто бывает полезным сопоста вить каждому узлу дерева метку или значение. Дерево, у которого узлам сопоставлены метки, называется ***помеченным деревом***. Метка узла – это значение, которое «хранится» в узле. Полезна следующая аналогия: дерево – список, узел – позиция, метка – элемент.

Рассмотрим пример дерева с метками, представляющее арифметическое выражение ( a + b )\*( a + c ), где n 1 , n 2 , …, n 7 – имена узлов, а метки проставлены рядом с соответствующими узлами. Правила соответствия меток деревьев элементам выражений следующие:

+

* Метка каждого листа соответствует операнду и содержит его значение;
* Метка каждого внутреннего (родительского) узла соответствует оператору.



Часто при обходе деревьев составляется список не имен узлов, а их меток. В случае дерева выражений при прямом обходе получим известную **префиксную форму** записи выражения, где оператор предшествует обоим операндам. В нашем примере мы получим префиксное выражение вида: \*+ ab + ac .

Обратный обход меток дерева дает **постфиксное представление** выражения (польскую запись). Обратный обход нашего дерева даст нам следующую запись выражения: ab + ac +\*.

Следует учесть, что префиксная и постфиксная запись выражения не требует скобок.

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

*Алгоритм построения помеченного дерева и вычисления выражений по нему.*

1. Предварительная обработка выражения (контроль правильности расстановки скобок в выражении, удаление пробелов, замена унарного минуса на (0-1)\*);

2. Построение дерева для данного выражения. Поиск в выражении арифметической операции с минимальным приоритетом с пропуском вложенных скобок. Разделение выражения на две части относительно найденной операции с наименьшим приоритетом и рекурсивное повторение процесса.

3. Вычисление выражения по построенному дереву.

На рисунке 13.6  приведен пример построения помеченного дерева по выражению. Выполняя заданный обход помеченного дерева, несложно вычислить значение исходного выражения.

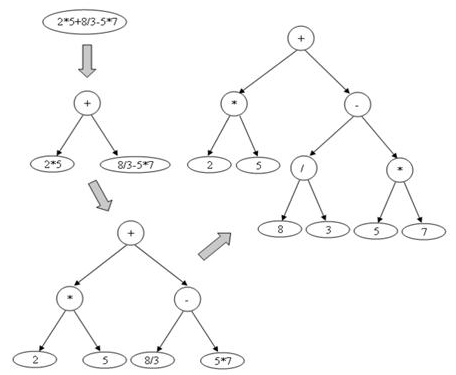


Рисунок 13.6 – Пример построения помеченного дерева

Часто при обходе деревьев составляется список не имен узлов, а их меток. В случае дерева выражений при прямом обходе получим известную **префиксную форму** записи выражения, где оператор предшествует обоим операндам. В нашем примере мы получим префиксное выражение вида: \*+ ab + ac .

Обратный обход меток дерева дает **постфиксное представление** выражения (польскую запись). Обратный обход нашего дерева даст нам следующую запись выражения: ab + ac +\*.

Следует учесть, что префиксная и постфиксная запись выражения не требует скобок.

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

Можно несколько сэкономить память компьютера за счет применения так называемых “прошитых” деревьев. В “прошитых” деревьях концевые связи-указатели используются для связи с родителями, такие связи назвали нитями. А для того чтобы отличать нормальные связи от нитей, в каждом узле хранят две однобитовые переменные LTag и RTag. Эти переменные равны нулю, если соответсвующие связи указывают на поддеревья, и единице, если связи являются нитями. Левая нить каждого узла указывает на узел, являющийся предшественником данного при центрированном обходе, правая - на узел, являющийся последователем данного узла.

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.

3.  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.

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

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

Также используются бинарные деревья, *прямо прошитые*справа и слева. В них пустые правые и левые указатели узлов заменены соответственно на их преемников и предшественников при прямом порядке просмотра. Прошитые деревья эффективно проходятся без использования стека.

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

При работе с древовидными структурами часто приходится решать

задачу обхода дерева. Это - способ методичного исследования узлов дерева,

при котором каждый узел проходится точно один раз. Для этого удобно

использовать, например, такой рекурсивный алгоритм:

1. Корень дерева.

2. Если нет поддеревьев, то ВЫХОД, иначе обходим все поддеревья

слева направо.

**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:

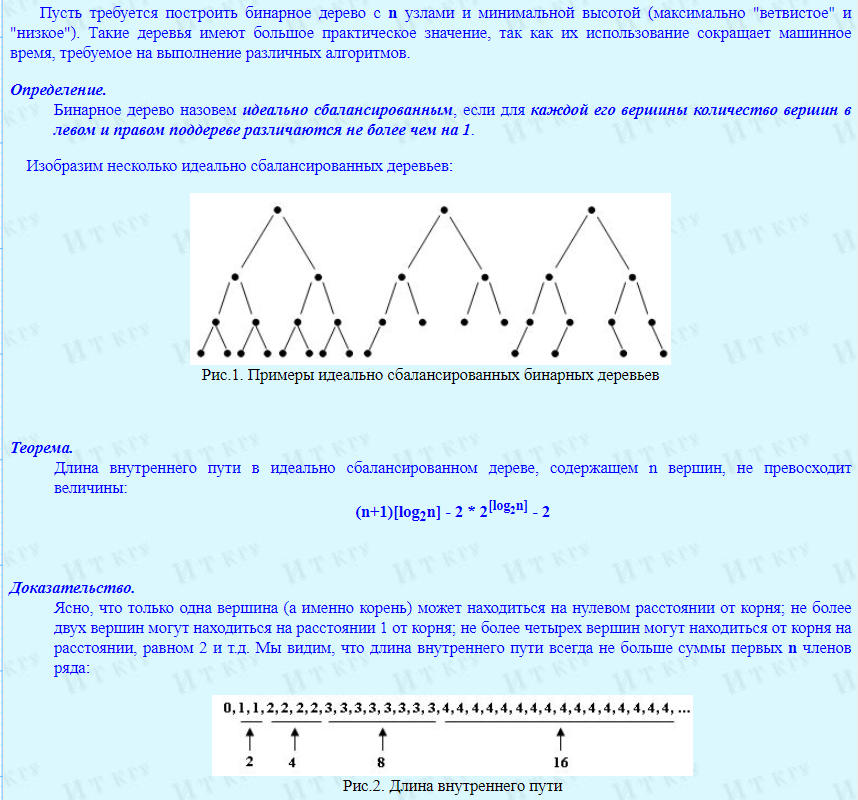
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Символ | А | Б | В | Г | Д |
| Частота | 15 | 7 | 6 | 6 | 5 |

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.

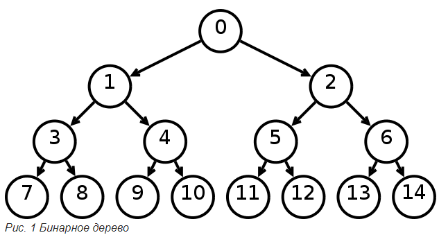
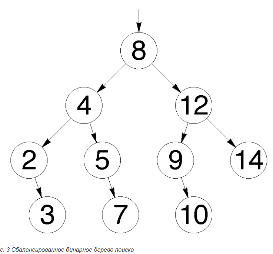
**42. Идеально сбалансированное бинарное дерево. Правила построения. Достоинства и недостатки. Привести пример такого дерева.**

****

**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.

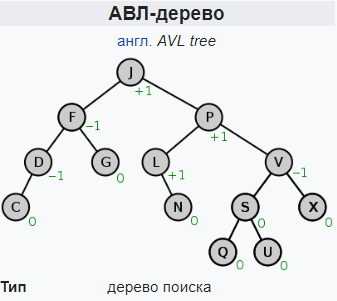
**Сбалансированное бинарное дерево поиска** — это бинарное дерево поиска с логарифмической высотой. Данное определение скорее идейное, чем строгое. В сбалансированном бинарном дереве поиска операции поиска, вставки и удаления выполняются за логарифмическое время (так как путь к любому листу от корня 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:

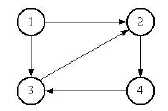
1. hl < hr: выравняется hl = hr. Ничего делать не нужно.
2. hl = hr: теперь левое поддерево будет больше на единицу, но балансировка пока не требуется.
3. hl > hr: теперь hl — hr = 2, — требуется балансировка.

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.

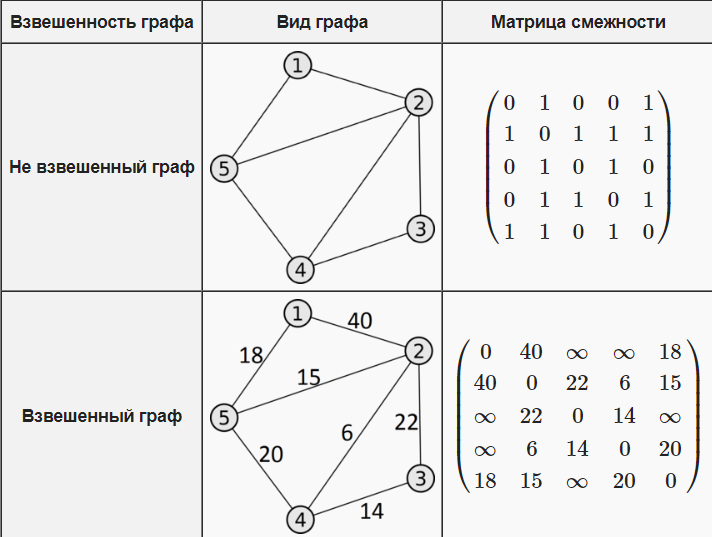
**Нерекурсивный алгоритм сложнее рекурсивного:** 1)Находится место вставки и вершина, высота которой не изменится при вставке (это вершина, у которой высота левого поддерева не равна высоте правого; будем называть её PrimeNode) 2) Выполняется спуск от PrimeNode до места вставки с изменением балансов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

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

**Орграф** – граф, ребрам которого присвоено направление. Направленные ребра называются **дугами**. **Вершины** называют узлами. Граф, ни одному ребру которого не присвоено направление, называется **неографом**. Вершины орграфа можно использовать для представления объектов, а дуги для отношений между объектами. **Путем** называется последовательность вершин, для которых существуют дуги. Путь начинается в 1ой вершине, проходит через все остальные по порядку следования и заканчивается в последней вершине. **Длина пути** – количество дуг, составляющих путь = n-1. Если у орграфа одна вершина, то путь=0. Путь называется **простым,** если все вершины на нем, кроме первой и последней, различны. Цикл – это простой путь длины не менее 1, который начинается и заканчивается в одной и той же вершине. **Помеченный орграф** - 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. **Цикл** в орграфе – простой путь длины не менее 1, который начинается и заканчивается в одной и той же вершине.

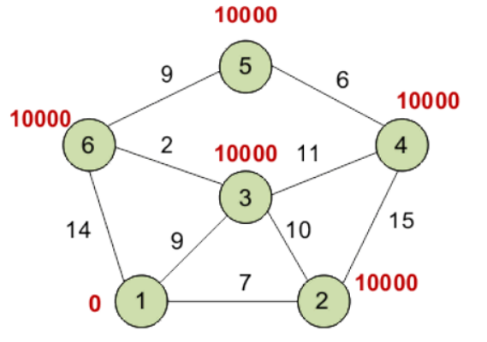


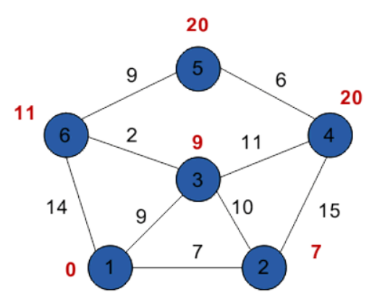
Для представления ориентированных графов можно использовать различные структуры данных. Выбор структуры данных зависит от операторов, которые будут применяться к вершинам и дугам орграфа. Чаще всего используют матрицу смежности. Предположим, что множество вершин орграфа V={1, 2, …, n}, тогда матрица смежности графа G – это матрица А размером n\*n со значениями булевого типа, где A[i,j]=true тогда и только тогда, когда существует дуга из вершины i в вершину j. Время доступа к элементам матрицы смежности зависит от размеров множества вершин и множества дуг. Представление орграфа в виде матрицы смежности удобно представлять в алгоритмах, где часто проверяется существование пути.



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

Алгоритм Дейкстры - алгоритм на графах, изобретённый нидерландским ученым Э. Дейкстрой в 1959 году. Находит кратчайшее расстояние от одной из вершин графа до всех остальных. Работает только для графов без рёбер отрицательного веса. Рядом с каждой вершиной красным обозначена метка – длина кратчайшего пути в эту вершину из вершины 1.



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 в 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. После шестого шага:

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.

Вывод 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. Для хранения весов графа используется квадратная матрица. В заголовках строк и столбцов находятся вершины графа. А веса дуг графа размещаются во внутренних ячейках таблицы. Граф не содержит петель, поэтому на главной диагонали матрицы содержатся нулевые значения.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** |
| **1** | 0 | 7 | 9 | 0 | 0 | 14 |
| **2** | 7 | 0 | 10 | 15 | 0 | 0 |
| **3** | 9 | 10 | 0 | 11 | 0 | 2 |
| **4** | 0 | 15 | 11 | 0 | 6 | 0 |
| **5** | 0 | 0 | 0 | 6 | 0 | 9 |
| **6** | 14 | 0 | 2 | 0 | 9 | 0 |

#define \_CRT\_SECURE\_NO\_WARNINGS  
#include <stdio.h>  
#include <stdlib.h>  
#define SIZE 6  
int main()  
{  
  int a[SIZE][SIZE]; // матрица связей  int d[SIZE]; // минимальное расстояние

  int v[SIZE]; // посещенные вершины

  int temp, minindex, min;  
  int begin\_index = 0;  
  system("chcp 1251");  
  system("cls");  
  // Инициализация матрицы связей

  for (int i = 0; i<SIZE; i++)  
  {  
    a[i][i] = 0;  
    for (int j = i + 1; j<SIZE; j++) {  
      printf("Введите расстояние %d - %d: ", i + 1, j + 1);  
      scanf("%d", &temp);  
      a[i][j] = temp;  
      a[j][i] = temp;  
    }  
  }  
  // Вывод матрицы связей

  for (int i = 0; i<SIZE; i++)  
  {  
    for (int j = 0; j<SIZE; j++)  
      printf("%5d ", a[i][j]);  
    printf("\n");  
  }  
  //Инициализация вершин и расстояний

  for (int i = 0; i<SIZE; i++)  
  {  
    d[i] = 10000;  
    v[i] = 1;  
  }  
  d[begin\_index] = 0;  
  // Шаг алгоритма

  do {  
    minindex = 10000;  
    min = 10000;  
    for (int i = 0; i<SIZE; i++)  
    { // Если вершину ещё не обошли и вес меньше min  
      if ((v[i] == 1) && (d[i]<min))  
      { // Переприсваиваем значени

        min = d[i];  
        minindex = i;  
      }  
    }  
    // Добавляем найденный минимальный вес

    // к текущему весу вершины

    // и сравниваем с текущим минимальным весом вершины

    if (minindex != 10000)  
    {  
      for (int i = 0; i<SIZE; i++)  
      {  
        if (a[minindex][i] > 0)  
        {  
          temp = min + a[minindex][i];  
          if (temp < d[i])  
          {  
            d[i] = temp;  
          }  
        }  
      }  
      v[minindex] = 0;  
    }  
  } while (minindex < 10000);  
  // Вывод кратчайших расстояний до вершин

  printf("\nКратчайшие расстояния до вершин: \n");  
  for (int i = 0; i<SIZE; i++)  
    printf("%5d ", d[i]);  
  // Восстановление пути

  int ver[SIZE]; // массив посещенных вершин  
  int end = 4; // индекс конечной вершины = 5 – 1

  ver[0] = end + 1; // начальный элемент - конечная вершина

  int k = 1; // индекс предыдущей вершины

  int weight = d[end]; // вес конечной вершины

  while (end != begin\_index) // пока не дошли до начальной вершины  
  {  
    for (int i = 0; i<SIZE; i++) // просматриваем все вершины

      if (a[end][i] != 0)   // если связь есть

      {  
        int temp = weight - a[end][i]; // определяем вес пути из

предыдущей вершины

        if (temp == d[i]) // если вес совпал с рассчитанным

        {                 // значит из этой вершины и был переход

          weight = temp; // сохраняем новый вес

          end = i;       // сохраняем предыдущую вершину

          ver[k] = i + 1; // и записываем ее в массив

          k++;  
        }  
      }  
  }  
  // Вывод пути (начальная вершина оказалась в конце массива из k элементов)

  printf("\nВывод кратчайшего пути\n");  
  for (int i = k - 1; i >= 0; i--)  
    printf("%3d ", ver[i]);  
  getchar(); getchar();  
  return 0;  
}

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

Алгоритм Флойда-Уоршелла используется для нахождения длины кратчайшего пути между всемипарами вершин во взвешенном графе за O(N3)O(N3).

В основе алгоритма лежит следующее: пусть dp[i][j][k] - длина кратчайшего пути между вершинами i и j, проходящего только через промежуточные вершины {0,1,…,k} (i и j не считаются).

Научимся пересчитывать пути при увеличении k: пусть мы хотим улучшить длину пути между i и j при увеличении k на 1. У нас есть два варианта: не использовать вершину k+1 (тогда dp[i][j][k+1]=dp[i][j][k]), или использовать, пройдя через неё. Утверждается, что кратчайший путь из i в k+1в j получается объединением кратчайших путей из i в k+1 и из k+1в j(используя промежуточные вершины до k, так как k+1 уже не является промежуточной). Тогда dp[i][j][k+1]=dp[i][k+1][k]+dp[k+1][j][k]. Сразу же попробуем сократить количество используемой памяти. Заметим, что для улучшения любого пути нам нужны только предыдущие значения длин путей для k−1. Можно просто откинуть последнее измерение (k) в массиве, приняв dp[i][j] - текущая минимальная длина пути из i в j, которую мы постепенно улучшаем. В качестве начальных значений, просто запишем, что кратчайший путь из каждой вершины в саму себя равен 0: dp[i][i]=0; кратчайший путь между двумя вершинами, между которыми есть ребро равен его длине: dp[i][j]=len(i,j); а кратчайший путь между всеми остальными парами вершин равен бесконечности: dp[i][j]=∞. Заметьте, что начальные значения не используют никаких промежуточных вершин, поэтому начинать работу алгоритма нужно с k=0. Для реализации нам не требуется хранить сам граф (в виде матрицы или списка смежности). Просто при вводе рёбер будем заполнять начальные значения. Затем по очереди используем каждую вершину от 0 до N−1 для улучшения пути между всеми парами вершин.

**#include <bits/stdc++.h>**

**using namespace std;**

**const int INF = 1e9 + 7;**

**int dp[1000][1000];**

**int main() {**

**int n, m;**

**cin >> n >> m;**

**for (int i = 0; i < n; i++) {**

**for (int j = 0; j < n; j++) {**

**dp[i][j] = INF;**

**}**

**}**

**for (int i = 0; i < n; i++) {**

**dp[i][i] = 0;**

**}**

**for (int i = 0; i < m; i++) {**

**int u, v, len;**

**cin >> u >> v >> len;**

**u--, v--;**

**dp[u][v] = dp[v][u] = len;**

**}**

**for (int k = 0; k < n; k++) {** //текущая вершина, используемая для улучшения

**for (int i = 0; i < n; i++) {**

**for (int j = 0; j < n; j++) {**

**dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);**

**}**

**}**

**}** //Массив dp содержит длины кратчайших путей между всеми парами вершин

**}**

**Поведение при наличии отрицательных циклов.**

В отличие от алгоритма Дейкстры, алгоритм Флойда-Уоршелла может корректно работать при наличии в графе рёбер с отрицательным весом. Для этого стоит немного изменить реализацию, добавив явную проверку на равенство длины пути бесконечности. Без неё в массиве могут появляться расстояния ∞−1, ∞−2, и т.д. Они могут вызвать проблемы при достаточно больших длинах рёбер, поэтому перепишем алгоритм следующим образом:

**for (int k = 0; k < n; k++) {**

**for (int i = 0; i < n; i++) {**

**for (int j = 0; j < n; j++) {**

**if (dp[i][k] < INF && dp[k][j] < INF) { //явная проверка**

**dp[i][j] = min(dp[i][j], dp[i][k] + dp[k][j]);**

**}**

**}**

**}**

**}**

Однако существуют графы, в которых отрицательный вес имеют не простые пути, а циклы. Они лишают смысла задачу о нахождении кратчайшего пути, так как позволяют получать пути бесконечно малой длины. С помощью алгоритма Флойда-Уоршелла можно легко проверять в графе наличие таких циклов: если после окончания работы алгоритма существует вершина i такая, что dp[i][i]<0d, то она входит в отрицательный цикл.

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

В ряде задач интерес представляет  только сам факт существования пути, длиной не меньше 1, от вершины *i* до вершины *j*. Для решения таких задач применяется алгоритм Уоршелла.

Предположим, что матрица стоимостей *С* совпадает с матрицей смежности для данного орграфа *G*, т.е. *C[i, j]*=1 только в том случае, если есть дуга , и *C[i, j]*=0, если такой дуги не существует. Требуется определить матрицу А такую, что *А[i, j]*=1 тогда и только тогда, когда существует путь от вершины *i* до вершины*j*длиной не менее 1 и  *А[i, j]*=0 – в противном случае. Такую матрицу *А* часто называют *транзитивным замыканием*матрицы смежности.

На рисунке 18.1 показан орграф, а на рисунке 18.2 – транзитивное замыкание матрицы смежности этого орграфа.

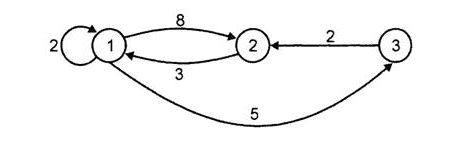


Рисунок 18.1 – Помеченный орграф

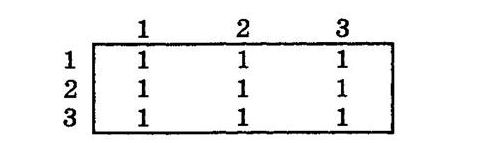


Рисунок 18.2 – Транзитивное замыкание матрицы смежности

       Транзитивное замыкание можно вычислить, применяя на *к*-ом шаге следующую формулу к булевой матрице *А*.

       Эта формула устанавливает, что существует путь от вершины *i* до вершины j, проходящих через вершины с номерами, не превышающими *k*, только в следующих случаях.

       1. Уже существует путь от вершины*i* до вершины *j*, который проходит через вершины с номерами, не превышающими *к*-1.

       2. Существует путь от вершины *i* до вершины *к*, проходящий через вершины с номерами, не превышающими *к-1*, и путь от вершины *к* до вершины *j*, который проходит через вершины с номерами, не превышающими *к*-1.

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

Определим понятие *центральной вершины*орграфа. Пусть v - произвольная вершина орграфа *G=(V, E)*. Эксцентриситет (максимальное удаление) вершины *v* определяется как *max{минимальная длина пути от вершины w до вершины v }.*

*Центром орграфа G* называется вершина с минимальным эксцентриситетом, т.е. это вершина, для которой максимальное расстояние (длина пути) до других вершин минимально.

Рассмотрим помеченный орграф, показанный на рис. 7.8.

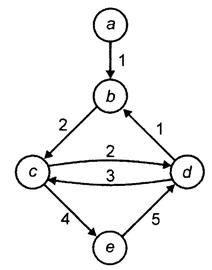


Рис. 7.8 – Помеченный орграф

В этом графе вершины имеют следующие эксцентриситеты.



Откуда видно, что центром данного орграфа является вершина *d*.

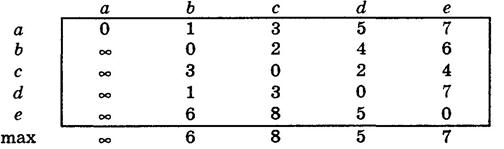
Пусть *С* – матрица стоимостей для орграфа *G*. Тогда центр орграфа можно найти, применив следующий алгоритм.

1. Применить алгоритм Флойда к матрице *С*для вычисления матрицы *А*, содержащей все кратчайшие пути орграфа *G*.

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

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

Матрица всех кратчайших путей для орграфа из рис. 7.8 представлена на рис. 7.9. Максимальные значения в каждом столбце приведены под матрицей.



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

Хеширование – распространенный метод обеспечения быстрого доступа к информации, хранящейся во вторичной памяти. Основная идея этого метода подобна открытому хешированию, рассмотренному ранее. Записи файла распределяются между *сегментами*, каждый из которых состоит из связного списка одного или нескольких блоков внешней памяти.

       Имеется таблица сегментов, содержащая *В* указателей,  –  по  одному  на каждый сегмент. Каждый указатель в таблице сегментов представляет собой физический адрес первого блока связного списка блоков для соответствующего сегмента. Сегменты пронумерованы от 1 до *В*. Хеш-функция *h* отображает каждое значение ключа в одно из целых чисел от 1 до *В*. Если *х* – ключ, то *h(x)* является номером сегмента, который содержит запись с ключом *х*, если такая запись вообще существует. Блоки, составляющие каждый сегмент, образуют связный список. Таким образом, заголовок *i*-ого блока содержит указатель на физический адрес (*i*+1)-ого блока. Последний блок сегмента содержит в своем заголовке *nil*-указатель. Такой способ организации показан на рисунке 19.1. При этом в данном случае элементы, хранящиеся в одном блоке сегмента, не требуется связывать друг с другом с помощью указателей, связывать между собой нужно только блоки.

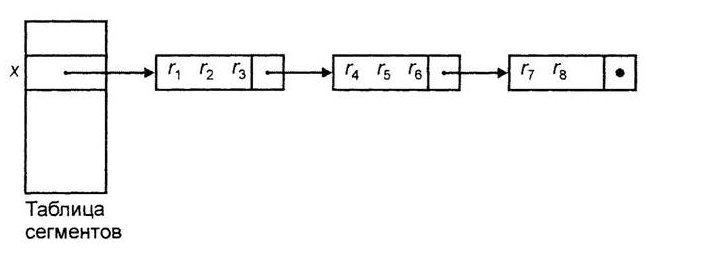


Рисунок 19.1 – Сегменты, состоящие из связанных блоков

         Если размер таблицы сегментов невелик, ее можно хранить в основной памяти, иначе ее можно хранить последовательным способом в отдельных блоках. Если требуется найти запись с ключом *х*, вычисляется *h(x)* и находится блок таблицы сегментов, содержащий указатель на первый блок сегмента *h(x)*. Затем последовательно считываются блоки сегмента *h(x)*, пока не обнаружится блок, содержащий запись с ключом *х*. Если исчерпаны все блоки в связном списке для сегмента *h(x)*, делается вывод, что *х* не является ключом ни одной из записей.

       Такая структура оказывается эффективной, если в выполняемом операторе указываются значения ключевых полей. Среднее количество обращений к блокам, требующееся для выполнения оператора, в котором указан ключ записи, приблизительно равняется среднему количеству блоков в сегменте, которое равно *n/bk*, если *n* – количество записей, блок содержит *b* записей, а *k* соответствует количеству сегментов. В результате, при такой организации данных операторы, использующие значения ключей, выполняются в среднем в *к* раз быстрее, чем в случае неорганизованного файла.

       Чтобы вставить запись с ключом, значение которого равняется *х*, нужно сначала проверить, нет ли в файле записи с таким значением ключа. Если такая запись есть, то выдается сообщение об ошибке, поскольку предполагается, что ключ уникальным образом идентифицирует каждую запись. Если записи с ключом *х* нет, новая запись вставляется в первый блок цепочки для сегмента *h(x)*, в который ее удается вставить. Если запись не удается вставить ни в один из существующих блоков сегмента *h(x)*, файловой системе выдается команда найти новый блок, в который будет помещена эта запись. Затем новый блок добавляется в конец цепочки блоков сегмента *h(x)*.

       Для удаления записи с ключом *х*, требуется сначала найти эту запись, а затем установить ее бит удаления.

       Удачная организация файлов с хешированным доступом требует лишь незначительного числа обращений к блокам при выполнении каждой операции с файлами. Если выбрана удачная функция хеширования, а количество сегментов приблизительно равно количеству записей в файле, деленному на количество записей, которые могут поместиться в одном блоке, тогда средний сегмент состоит из одного блока. Если не учитывать обращения к блокам, которые требуются для просмотра таблицы сегментов, типичная операция поиска данных, основанного на ключах, потребует одного обращения к блоку, а операция вставки, удаления или изменения потребуют двух обращений к блокам. Если среднее количество записей в сегменте намного превосходит количество записей, которые могут поместиться в одном блоке, можно периодически реорганизовывать таблицу сегментов, удваивая количество сегментов и деля каждый сегмент на две части.

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

 Еще одним распространенным способом организации файла записей является поддержание файла в отсортированном по значениям ключей порядке. В этом случае файл можно просматривать как словарь или телефонный справочник, когда просматриваются лишь заглавные слова или фамилии на каждой странице. Чтобы облегчить процедуру поиска, можно создать второй файл, называемый *разреженным индексом*, который состоит из пар *(x, b)*, где *x*– значение ключа, а *b* – физический адрес блока, в котором значение ключа первой записи равняется *х*. Разреженный индекс отсортирован по значениям ключей.

       На рисунке 19.2 показан файл с информацией и соответствующий ему файл разреженного индекса.

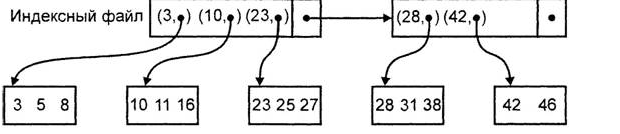


Рисунок 19.2 – Файл и его разреженный индекс

       Предполагается, что три записи основного файла или три пары индексного файла помещаются в один блок. Записи основного файла представлены только значениями ключей, которые в данном случае являются целочисленными величинами. Чтобы найти запись с заданным ключом *х*, нужно сначала просмотреть индексный файл, отыскивая в нем пару *(x, b)*. В действительности ищется наибольшее *z*, такое, что  и далее находится пара (*z, b)*. В этом случае ключ*х* оказывается в блоке *b*, если такой ключ вообще присутствует в основном файле.

       Чтобы создать индексированный файл, записи сортируются по значениям их ключей, а затем распределяются по блокам в возрастающем порядке ключей. В каждый блок можно поместить или столько записей, сколько туда помещается, или оставить в нем вакантные места с возможностью добавления записей впоследствии. Преимущества такого подхода заключаются в том, что вероятность переполнения блока, куда вставляются новые записи, в этом случае оказывается ниже, иначе нужно будет обращаться к смежным блокам. После распределения записей по блокам создается индексный файл: просматривается по очереди каждый блок и находится первый ключ в каждом блоке. Подобно тому, как это сделано в основном файле, в блоках, содержащих индексный файл, можно оставить какое-то место для последующего роста.

       Допустим есть отсортированный файл записей, хранящихся в блоках  Для вставки новой записи в отсортированный файл используем индексный файл, с помощью которого определяется, блок с каким номером должен содержать новую запись. Если новая запись помещается в блок  она туда заносится в корректной последовательности. Если новая запись становится первой записью в блоке  тогда выполняется корректировка индексного файла.

       Если новая запись не помещается в блок  можно применить следующую стратегию. Перейти на блок  и узнать, можно ли последнюю запись  переместить в начало  Если можно, последняя запись перемещается в  а новую запись можно затем вставить на подходящее место в  В этом случае нужно откорректировать вход индексного файла для  и, возможно, для

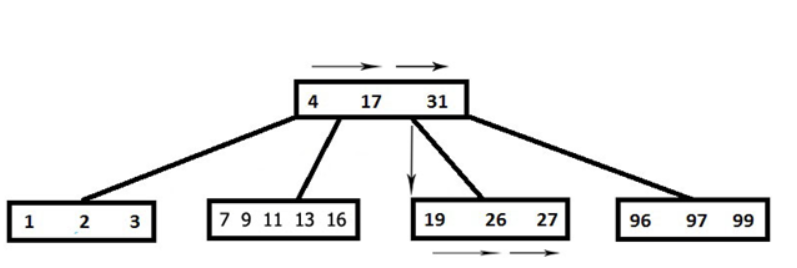
       Если блок  также заполнен или если  является последним блоком (*i=m*), из файловой системы нужно получить новый блок. Новая запись вставляется в этот новый блок, который должен размещаться вслед за блоком  Затем используется процедура вставки в индексном файле записи для нового блока.

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

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

**Поиск записей.** Поиск в B-дереве очень схож с поиском в бинарном дереве, только здесь мы должны сделать выбор пути к потомку не из 2 вариантов, а из нескольких. В остальном — никаких отличий. На рисунке ниже показан поиск ключа 27. Поясним иллюстрацию (и соответственно стандартный алгоритм поиска): 1) Идем по ключам корня, пока меньше необходимого. В данном случае дошли до 31. 2) Спускаемся к ребенку, который находится левее этого ключа. 3) Идем по ключам нового узла, пока меньше 27. В данном случае – нашли 27 и остановились.

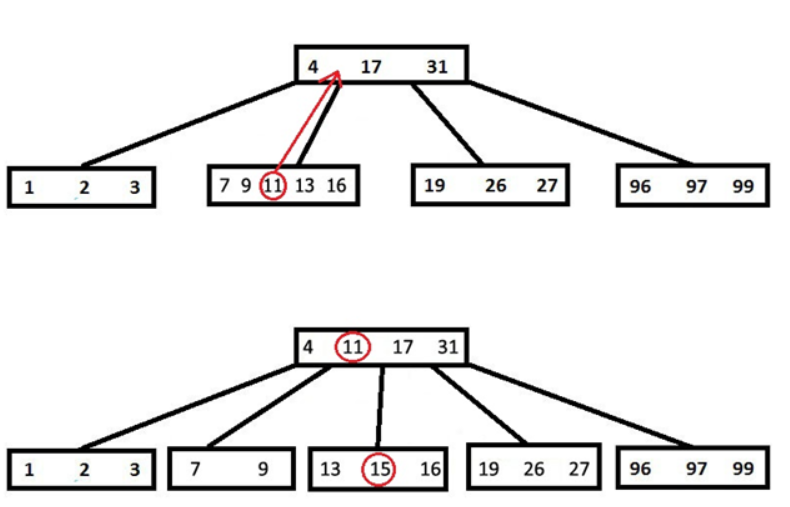
**Пример** **поиска:**



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

**Вставка записей.**

В отличие от поиска, операция добавления существенно сложнее, чем в бинарном дереве, так как просто создать новый лист и вставить туда ключ нельзя, поскольку будут нарушаться свойства B-дерева. Также вставить ключ в уже заполненный лист невозможно => необходима операция разбиения узла на 2. Если лист был заполнен, то в нем находилось 2t-1 ключей => разбиваем на 2 по t-1, а средний элемент (для которого t-1 первых ключей меньше его, а t-1 последних больше) перемещается в родительский узел. Соответственно, если родительский узел также был заполнен – то нам опять приходится разбивать. И так далее до корня (если разбивается корень – то появляется новый корень и глубина дерева увеличивается). Как и в случае обычных бинарных деревьев, вставка осуществляется за один проход от корня к листу. На каждой итерации (в поисках позиции для нового ключа – от корня к листу) мы разбиваем все заполненные узлы, через которые проходим (в том числе лист). Таким образом, если в результате для вставки потребуется разбить какой-то узел – мы уверены в том, что его родитель не заполнен! **Пример**:

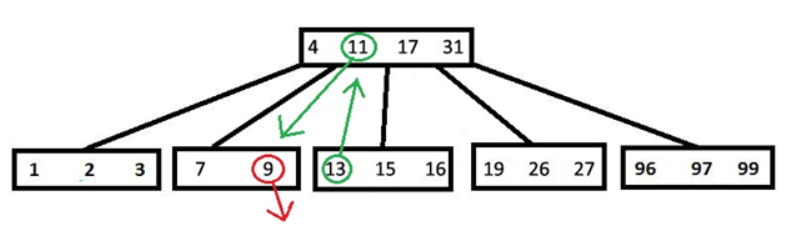
Добавляем ключ «15». В поисках позиции для нового ключа мы натыкаемся на заполненный узел (7, 9, 11, 13, 16). Следуя алгоритму, разбиваем его – при этом «11» переходит в родительский узел, а исходный разбивается на 2. Далее ключ «15» вставляется во второй «отколовшийся» узел. Все свойства B-дерева сохраняются!

Операция добавления происходит также за время O(t logt n). Важно опять же, что дисковых операций мы выполняем всего лишь O(h), где h – высота дерева.

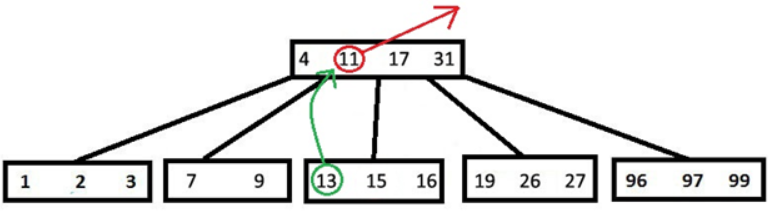
**55. Удаление заданного элемента из В-дерева.**

**Удаление записей.**

Удаление ключа из B-дерева еще более громоздкий и сложный процесс, чем вставка. Это связано с тем, что удаление из внутреннего узла требует перестройки дерева в целом. Аналогично вставке необходимо проверять, что мы сохраняем свойства B-дерева, только в данном случае нужно отслеживать, когда ключей t-1 (то есть, если из этого узла удалить ключ – то узел не сможет существовать). Рассмотрим алгоритм удаления:  
1)Если удаление происходит из листа, то необходимо проверить, сколько ключей находится в нем. Если больше t-1, то просто удаляем и больше ничего делать не нужно. Иначе, если существует соседний лист (находящийся рядом с ним и имеющий такого же родителя), который содержит больше t-1 ключа, то выберем ключ из этого соседа, который является разделителем между оставшимися ключами узла-соседа и исходного узла (то есть не больше всех из одной группы и не меньше всех из другой). Пусть это ключ k1. Выберем ключ k2 из узла-родителя, который является разделителем исходного узла и его соседа, который мы выбрали ранее. Удалим из исходного узла нужный ключ (который необходимо было удалить), спустим k2 в этот узел, а вместо k2 в узле-родителе поставим k1. Чтобы было понятнее ниже представлен рисунок (рис.1), где удаляется ключ «9». Если же все соседи нашего узла имеют по t-1 ключу. То мы объединяем его с каким-либо соседом, удаляем нужный ключ. И тот ключ из узла-родителя, который был разделителем для этих двух «бывших» соседей, переместим в наш новообразовавшийся узел (очевидно, он будет в нем медианой).



2)Теперь рассмотрим удаление из внутреннего узла x ключа k. Если дочерний узел, предшествующий ключу k содержит больше t-1 ключа, то находим k1 – предшественника k в поддереве этого узла. Удаляем его (рекурсивно запускаем наш алгоритм). Заменяем k в исходном узле на k1. Проделываем аналогичную работу, если дочерний узел, следующий за ключом k, имеет больше t-1 ключа. Если оба (следующий и предшествующий дочерние узлы) имеют по t-1 ключу, то объединяем этих детей, переносим в них k, а далее удаляем k из нового узла (рекурсивно запускаем наш алгоритм). Если сливаются 2 последних потомка корня – то они становятся корнем, а предыдущий корень освобождается. Ниже представлен рисунок (рис.2), где из корня удаляется «11» (случай, когда у следующего узла больше t-1 ребенка).



Операция удаления происходит за такое же время, что и вставка O(t logt n). Да и дисковых операций требуется всего лишь O(h), где h – высота дерева.