Оглавление

1. Общее

Абстрактные структуры данных – это модели связанных данных и строго определенный набор операций с ними.

Абстрактные структуры данных программируют на основе базовых структур данных. Использование абстрактных структур обусловлено огромным количеством разработанных вычислительных алгоритмов и алгоритмов обработки данных с доказанной эффективностью. Модели вычислений этих алгоритмов построены на классических абстрактных структурах данных.

К абстрактным структурам относятся стеки, очереди, деревья, графы, мультисписки, линейные и нелинейные списки и др. Отличие абстрактных структур друг от друга определяется набором операций и моделью связей между элементами, узлами структуры.

Одну и ту же абстрактную структуру можно программировать как на основе массива, так и на динамическом связном списке или на комбинации этих структур. Выбор в пользу той или иной базовой структуры определяют требования задачи к списку операций – алгоритмов, которыми эффективнее обрабатывать данные для решения.

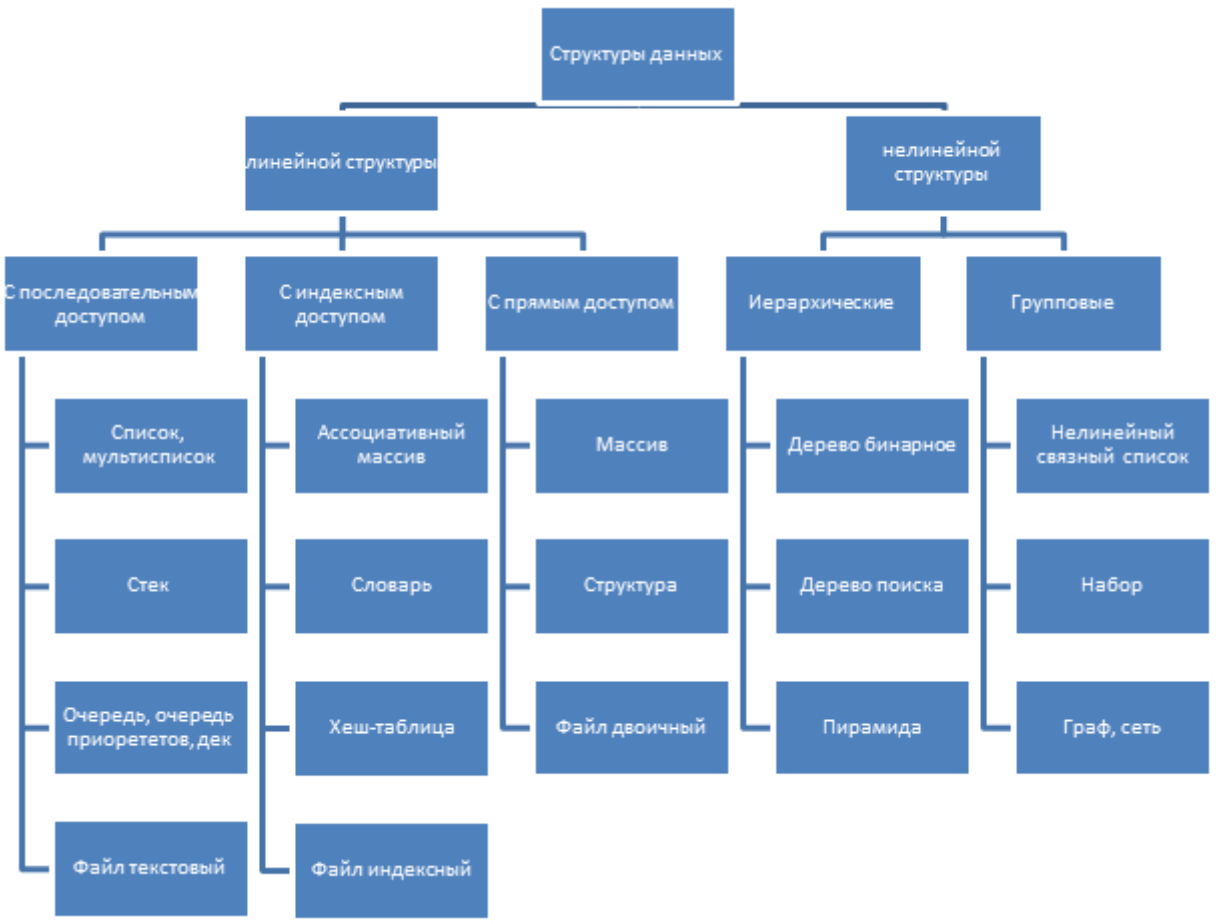


Рис.32 Классификация абстрактных структур данных.

Классифицируем структуры данных в зависимости от линейности/нелинейности структуры и по критерию доступа к элементам линейной структуры, рис 32. *Доступ к элементу* определяет построение алгоритма обхода элементов.

Последовательный доступ – это доступ к n-му элементу последовательности путем прохода через n-1 элемент.

Прямой доступ к элементу предполагает доступ по указанию позиции расположения элемента.

Индексный доступ предполагает наличие пары «ключ-элемент», где ключ является индексом и по индексу осуществляется прямой доступ к элементу. Основная операция структур с индексным доступом – это эффективный поиск.

Основные определения

Список – это последовательная организация данных, характеризуется отсутствием ограничений в наборе операций, является и базовой и абстрактной структурой.

Стек – это линейная структура данных, список данных, где операции включения и исключения элементов ограничены только одним концом структуры, реализован принцип «последний пришел - первый ушел».

Очередь – это линейная структура данных, список данных, где операции включения и исключения элементов выполняются с разных концов структуры, реализован принцип «первый пришел - первый ушел».

Очередь приоритетов – это очередь, где операция исключения реализована для элемента с наивысшим приоритетом.

Дек – это очередь, где операции включения и исключения элементов выполняются с обоих концов структуры.

Ассоциативный массив — это абстрактная структура данных, которая состоит из набора пар «ключ-значение», называемых ***ассоциациями***. Доступ к значению элемента прямой, по индексу ключа. Основными операциями будут включение, исключение пары и поиск по ключу. Ассоциативный массив не может хранить две пары с одинаковыми ключами.

Словарь – это ассоциативный массив, где разрешено хранение пар с одинаковыми ключами. Основными операциями будут включение, исключение и поиск пар, реализованные с учетом повторяющихся ключей.

Хеш-таблица – это структура данных на основе ассоциаций, пар «ключ-значения», но ключ преобразуется в индекс по заданной таблице. Таким образом, хеш-таблица – интерфейс ключа в ассоциативном массиве. Основные операции включения, исключения пар и поиск по ключу.

Массив – это последовательность *однотипных* элементов, где к элементу определен прямой доступ по индексу позиции (целое значение).

Структура (запись) – это последовательность *разнотипных* элементов, где к элементу определен прямой доступ, может быть определена как тип данных языков программирования.

Файл – это списковая структура данных с различным доступом к элементам, расположенная на внешних носителях компьютера, определена как тип данных языков программирования.

Иерархическая структура - древовидная структура данных, где есть разделение элементов по уровням. Элемент данного уровня может иметь дочерние элементы на следующем уровне.

Дерево – это иерархическая структура данных, где все элементы расположены по уровням и происходят из одного родительского элемента – корня дерева, он расположен на нулевом уровне. Каждый элемент нижнего уровня имеет ровно одного предка. Основные операции включение, исключение, обход и поиск.

Пирамида, двоичная куча – это сортирующее дерево, у которого минимальный/максимальный элемент расположен в корне.

Множество – неупорядоченная структура данных, которая позволяет хранить только уникальные значения элементов.

Граф – нелинейная структура данных, заданная описанием элементов вершин графа и элементов связей между вершинами.

Сеть – это граф, где задан вес каждой связи.

Основные операции абстрактных структур данных:

§ обход элементов структуры данных: доступ к каждому элементу для обработки;

§ добавление и/или вставка элементов в структуру данных;

§ удаление элемента структуры данных;

§ поиск элемента в структуре данных;

§ создание и разрушение структуры данных;

§ проверка на наличие элементов в структуре (проверка на пустоту).

Методика программирование абстрактных структур данных:

1. Выбирают оптимальную базовую структуру данных для реализации.
2. Определяют переменные памяти и разрабатывают функции алгоритмов операций данной абстрактной структуры.
3. Описывают абстрактный тип данных (АТД) для структуры.
4. Создают объект АТД и решают поставленную задачу.

Таблица 2

Сравнительные возможности базовых структур данных

Массив	Динамический связный список
Память выделяется единым непрерывным блоком под весь объявленный массив перед его использованием.	Память выделяется по мере добавления элементов.
Операции добавления/удаления элементов требуют перемещения всех элементов путем сдвига.	Операции добавления/удаления требуют переопределить указатели соседних элементов и выделения памяти.

Объем памяти выделяют с дополнительным резервом для добавляемых элементов.	Объем памяти для хранения соответствует количеству элементов в списке.
Прямой доступ к элементам	Последовательный доступ к элементам

Если операции добавления и удаления являются частыми, то правильно будет выбрать связный список, иначе удобно программировать на массиве.

2. Стек

Стек – это линейная структура данных, список данных, где операции включения и исключения элементов ограничены только одним концом структуры, реализован принцип «последний пришел - первый ушел». Принцип обозначается аббревиатурой LIFO (last in-first out) и другое название стека – магазин.

Стек (Stack) применяют как структуру данных, если в обработке необходим доступ только к первому элементу списка. Этот элемент в стеке называю *вершиной* стека и обозначают обычно переменной *top,* см. рис 33.

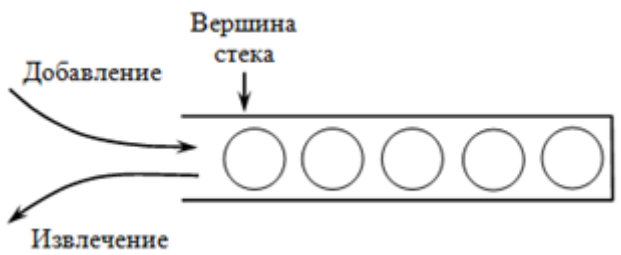


Рис.33 Графическая модель стека

Операции стека и их классическое обозначение, см. рис.34:

§ добавление элемента в вершину стека - push(val),

§ удаление/извлечение элемента из стека (с выдачей значения удаляемого элемента) – < тип элемента> pop().

§ прочитать значение верхнего элемента (если стек не пуст) – peek(),

§ проверить, пуст ли стек - isEmpty(),

§ очистка стека – clear(),

§ вывод размера стека- count().

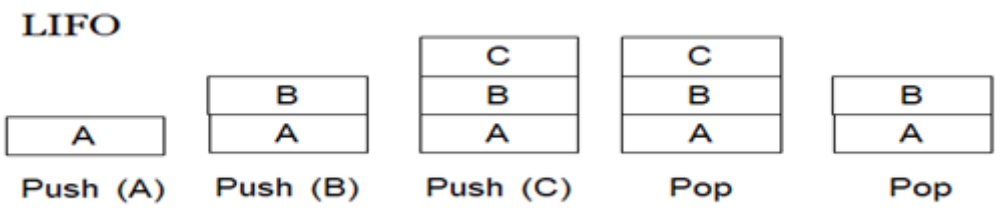


Рис.34 Графическая модель операций стека

Программирование стека возможно на базовых структурах одномерный массив и динамический односвязный список.

Программирование стека на массиве характеризуется:

- ограничением максимального размера стека размером массива, что порождает ошибку переполнения стека;

- простые алгоритмы реализации операций.

Программирование стека на списке характеризуется:

- нет проблемы переполнения стека, но есть ограничение по размеру динамической памяти - кучи;

- последовательный доступ к элементам, что не критично для стека, но усложняет.

2.1. Реализация стека одномерным массивом

Реализация стека одномерным массивом

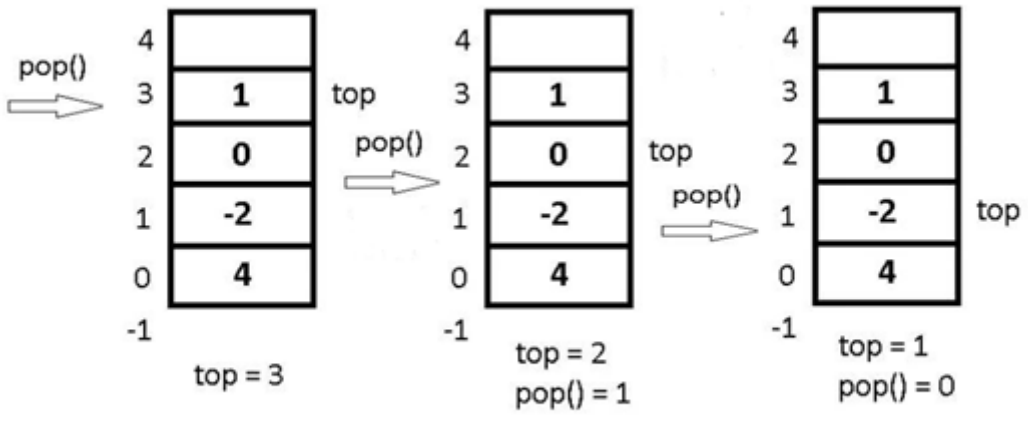


Рис.35 Реализация стека на массиве

1. Объявить массив S[MaxSize], для хранения элементов стека.

2. Объявить переменную top целого типа, для хранения индекса вершины стека – указатель на вершину. Пока стек пуст top=-1, иначе элементы стека будут расположены в диапазоне индексов массива от S[0] до S[top], рис 35.

3. Разработать функции операций со стеком.

Программа 2.1 Функции простейшей реализации операций стека на массиве.

```
//-------проверка на пустоту---------------------------------------

bool isEmpty() {

 if ( top==-1)

       return true;

 else

      return false;

}

//-------------------вставка элемента------------------------------

void push(int val){

 top++;

 S[top]=val;

}

//-------------------удаление элемента------------------------------

int pop(){

 if (isEmpty()) return -1;

  top--;

 return S[top+1];

}

//----------------------------------------------------------------
```

Реализация стека **на основе описания АТД** (абстрактного типа данных).

Описания типа данных Stack:

```
const int MaxSize=100;

struct Stack{

  int S[MaxSize];              // массив под стек

  int top=-1;                  // указатель вершины

};
```

Программа 2.2 Функции операций стека на основе типа данных Stack.

```
#include <iostream>
#include <iomanip>
#include <stdio.h>
using namespace std;

const int MaxSize = 100;
struct Stack {
    int S[MaxSize];                    // массив под стек
    int top = -1;                      // указатель вершины
};

//-------проверка на пустоту-----------------------------------------
bool isEmpty(Stack &St) {
    if (St.top == -1)
        return true;
    else
        return false;
}
//----------------добавление элемента-----------------------------
void push(Stack &St, int val) {
    if (St.top == MaxSize - 1) return;
    St.top++;
    St.S[St.top] = val;
}
//------------------удаление элемента-----------------------------
int pop(Stack &St) {
    if (isEmpty(St)) return -1;
    St.top--;
    return St.S[St.top + 1];
}
//---------------проверка-----------------------------------------
int main()
{
    setlocale(LC_ALL, "Russian");

    Stack S;        //объект
    push(S, 25); push(S, 35); push(S, 45);
    while (!isEmpty(S))
        cout << setw(5) << pop(S)<< endl;
```

2.2. Реализация стека односвязным динамическим списком

Реализация стека односвязным динамическим списком

Для программирования стека нельзя использовать весь набор алгоритмов обработки односвязного списка, только те, что соответствуют операциям стека. Алгоритмы односвязного списка, которые используют для реализации операций стека:

§ Функция addHead() - push();

§ Функция removeHead() – pop();

§ Функция retrieve() – peek();

§ Функция isEmpty() - isEmpty(),

§ Функция removeAll() - очистка стека – clear(),

§ Функция count() списка - вывод размера стека- count().

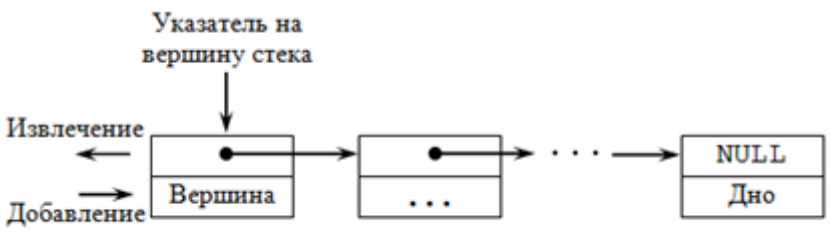


Рис.36 Реализация стека на динамическом связном списке

Описания типа данных элемента стека – это описание узла списка:

```
struct  Node {
  int  data;
  Node *prev;
};
Node* top=NULL;
```

Программа 2.3 Функции операций стека на основе динамического односвязного списка.

```
//------- Добавление элемента в стек --------------------------------
Node* push (Node  *top, int val){
    Node *temp = new Node;
    temp->prev= top;
    temp->data = val;
    top = temp;
 return top;
}
//------- Удаление элемента из стека---------------------------------
int pop(Node *&top){
 Node  *temp;
 int  Value=-1;
 if (top) {
       Value=top->data;
        temp=top->prev;
```

```
        delete top;
        top=temp;
      }
  return Value;
}
//------- Показать значение верхнего элемента-----------------------
int peek (Node*  top){
  if (top)
      return top->data;
  else return -1;
}
//-------Проверка на пустоту----------------------------------------
bool isEmpty(Node* top) { return !top; }
//----------------------------------------------------------------
int main(){
    top = push(top, 25);
    top = push(top, 35);
    top = push(top, 45);
    while (!isEmpty(top)) cout << setw(5) << pop(top);
}
//----------------------------------------------------------------
```

При реализации стека надо предусмотреть обработку двух исключительных ситуаций:

- при добавлении элемента контролировать переполнение массива или динамической памяти;

- при удалении элемента контролировать наличие элементов в стеке.

3. Очередь

Очередь – это линейная структура данных, список данных, где операции включения и исключения элементов выполняются с разных концов структуры, реализован принцип «первый пришел - первый ушел». Принцип обозначается аббревиатурой FIFO (first in-first out). Примером может служить обычная очередь в кассу.

Очередь (Queue) определяется двумя указателями на начало и конец, их называют *головой (head) и хвостом (tail)* очереди. Голова указывает на первый элемент очереди и используется для операции извлечения элемента, а хвост указывает на последний элемент очереди и используется для операции добавления в очередь элемента, см. рис 37.

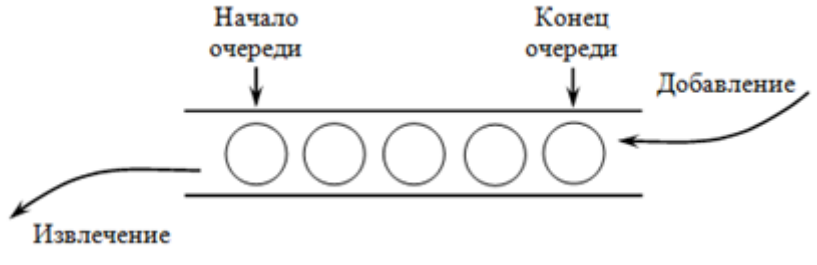


Рис.37 Графическая модель очереди

Операции очереди:

§ добавление элемента в конец tail очереди - enqueue (val);

§ удаление/извлечение элемента из начала head очереди (с выдачей значения удаляемого элемента) – < тип элемента> dequeue();

§ прочитать значение начала и конца очереди (если не пуста) – peekHead(), peekTail();

§ проверить, пуста ли очередь - isEmpty();

§ очистка очереди – clear();

§ вывод размера очереди - count(), length().

Очередь реализуют:

на основе статического или динамического одномерного массива;

на динамическом связном списке.

Реализация очереди на одномерном массиве

Два способа реализации:

§ *На одномерном массиве путем операции сдвига* к началу по мере удаления элементов из очереди и контроле переполнения в процессе добавления элементов в очередь. Происходит контроль указателя начала (head) и указателя конца, где они либо увеличиваются операцией добавления, либо уменьшается операцией удаления элемента из очереди. В процессе контроля, по мере необходимости выполняют сдвиг. Если реализация происходит на динамическом массиве, то по мере добавления можно расширять массив. Алгоритм работы на рис. 38.

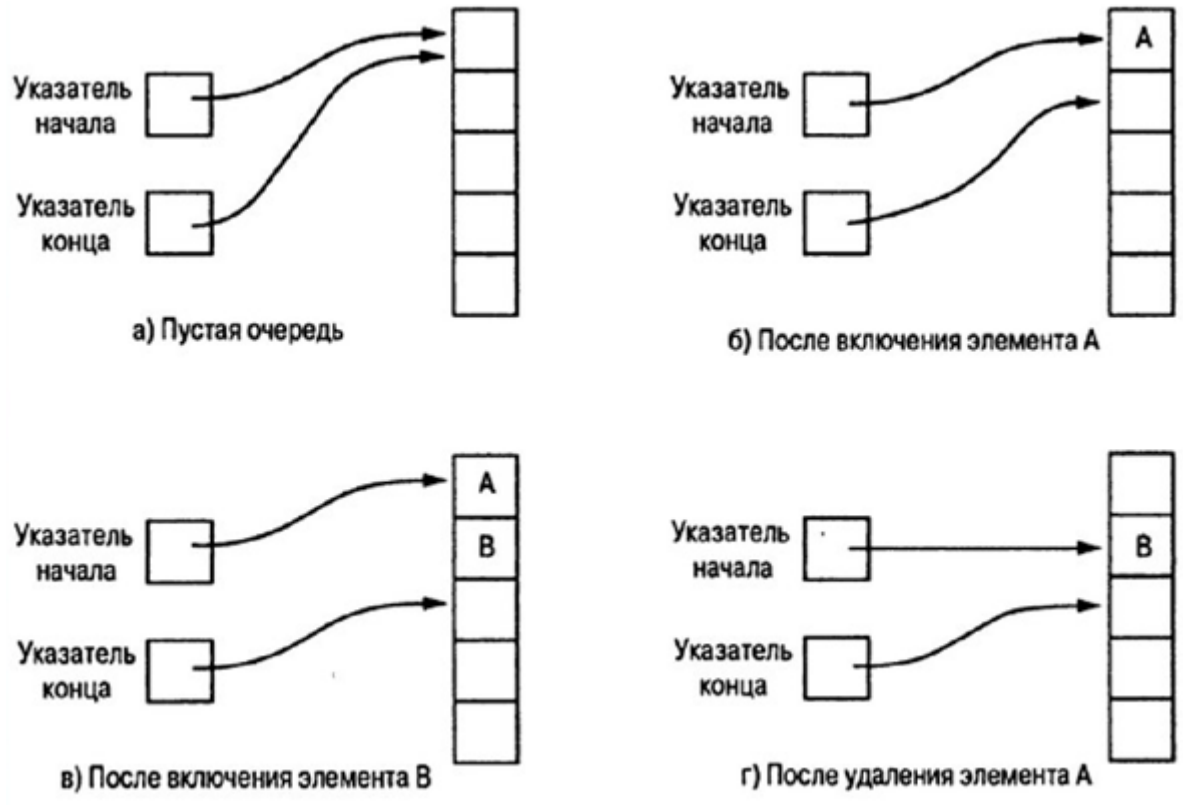


Рис.38 Реализация очереди на одномерном массиве

§ *На кольцевом (циклическом) одномерном массиве*, где в ограниченном размере массива движения указателей происходить по кольцу по мере добавления элементов и удаления (рис. 39). Концептуально кольцевой массив можно представить как массив, где после последнего элемента следует первый.

Исключительная ситуации: переполнение массива.

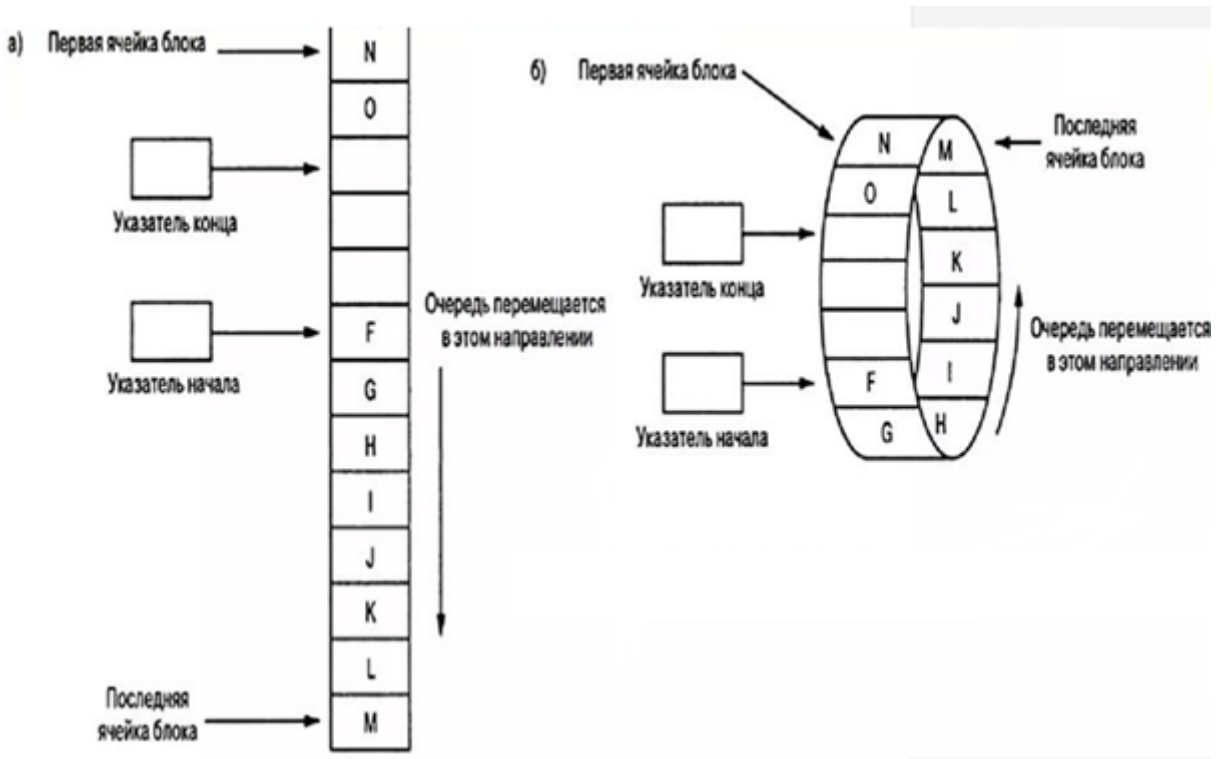


Рис.39 Реализация очереди на одномерном кольцевом массиве

Реализация простейшей очереди на базе массива:

1. В качестве базы выступает массив Q фиксированной длины N, таким образом, очередь ограничена и не может содержать более N-1 элементов.

2. Индексы элементов массива изменяются в пределах от 1 до N.

3. Реализация очереди хранит две простые переменные: индекс начала очереди (голова – head), индекс конца очереди (хвост – tail).

4. Первоначально имеем head=0, tail=0, т.е. очередь пуста.

Псевдокоды основных операций на кольцевом массиве см. на рис. 40.

Программа 2.4 Функции простейшей реализации операций очереди на кольцевом массиве.

```
//------------------вставка элемента------------------------------

void enqueue(int val){
```

```
 Q[tail]=val;
 if (tail== N-1)                    //проверка конца массива
     tail=0;
 else tail++;
}
//------------------удаление элемента------------------------------
int dequeue (){
  int val= Q[head];
  if (head== N-1)
       head=0;
   else head++;
 return val;
}
//-------проверка на пустоту-----------------------------------
bool isEmpty() {
 if ( tail==head)
        return true;
 else
       return false;
}
//-------проверка----------------------------------------------
int main(){
    enqueue(25); enqueue(35); enqueue(45);
    while (!isEmpty()) cout << setw(5) << dequeue();
}
//-----------------------------------------------------------------
```

```
ENQUEUE(Q, x)
1  Q[tail[Q]] ← x
2  if tail[Q] = length[Q]
3      then tail[Q] ← 1
4      else tail[Q] ← tail[Q] + 1

DEQUEUE(Q)
1  x ← Q[head[Q]]
2  if head[Q] = length[Q]
3      then head[Q] ← 1
4      else head[Q] ← head[Q] + 1
5  return x
```

Рис.40 Псевдокоды основных операций очереди на кольцевом массиве (Кормен).

Реализация очереди на динамическом кольцевом массиве как АТД (абстрактного типа данных). Описания типа данных Queue:

```
struct Queue{
  int head;              //указатель начала
```

```
  int tail;            // указатель конца

  int size;            //размер очереди

  int *Q;              //динамический массив под очередь

};
```

Программа 2.5 Функции операций очереди на типе АТД Queue.

```
//------выделение памяти под объект и инициализация полей типа------
      Queue * NewQueue(int size){
           return new Queue{ 0, 0, size, new int[size] };
      }
//------добавляем элемент в очередь---------------------------------
     void Enqueue(Queue* q, int val){


           q->Q[q->tail++] = val;
           if (q->tail == q->size)
                q->tail = 0;
      }
 //------удаляем элемент из очереди----------------------------------
      int Dequeue(Queue *q){
           int val = q->Q[q->head++];
           if (q->head == q->size)
                q->head = 0;
           return val;
     }
//-------вывести значение начала и хвоста очереди-------------------
     int PeekHead(Queue *q){
           return q->Q[q->head];
     }

     int PeekTail(Queue *q){
           return q->Q[q->tail];
     }
//--------очистить очередь ----------------------------------------
     void Clear(Queue *q){
           q->head = 0;
           q->tail = 0;
     }
//--------освободить память----------------------------------------
     void DeleteQueue(Queue *q){
           delete[] q->Q;
           delete q;
     }
//-----------------------------------------------------------------
```

3.1. Реализация очереди на базе связного динамического списка

Для реализации очереди могут быть использованы все виды динамических списков как линейных, так и кольцевых. Модель очереди на двусвязном списке представлена на рис 41.

Описания типа данных элемента очередь (рис. 41) – описание узла двусвязного списка:

```
struct Node{
   int data;
   Node* next;
   Node* prev;
};
Node* head=NULL; Node* tail=NULL;
```

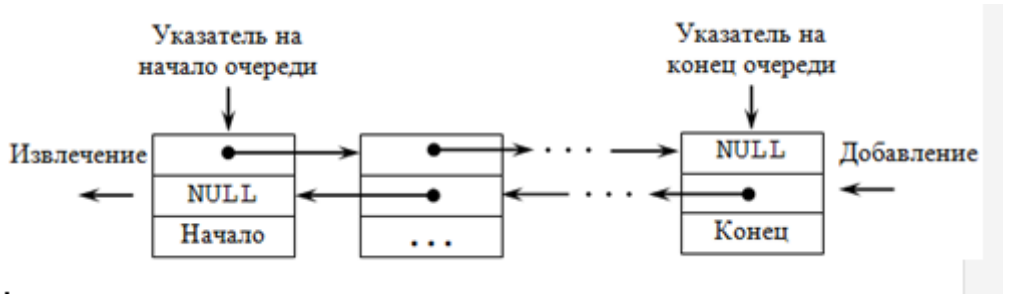


Рис.41 Модель очереди на двусвязном динамическом списке

Для программирования очереди нельзя использовать весь набор алгоритмов обработки связного списка, используют только те, что соответствуют операциям очереди. Алгоритмы связного списка, которые используют для реализации операций очереди:

§ Функция addTail() - enqueue ();

§ Функция removeHead() – dequeue();

§ Функция retrieve(head), retrieve(tail) – peekHead(), peekTail();

§ Функция isEmpty() - isEmpty(),

§ Функция removeAll() - очистка очереди – clear(),

§ Функция count() списка - вывод размера очереди- count().

Программа 2.6 Функции отдельных операций очереди на основе динамического односвязного списка.

```
//--------описание данных-----------------------------------------
      struct Node{
            int data;
            Node* next;
      };
      Node* head=NULL; Node* tail=NULL;
//------ добавление элемента в очередь-----------------------------
      void  enqueue(Node *&head, Node *&tail,  int val){
       Node *temp=new Node;
       temp->data=val;
       temp-> next = NULL;
       if (head==NULL) head=temp;
                  else tail->next=temp;
       tail=temp;
      }
//------ удаление элемента из очереди-----------------------------
      int dequeue(Node *&head, Node *&tail)   {
        Node *temp;   int Value=-1;
        if (head==tail) tail=NULL;
        if (head) {
             Value=head->data;
             temp=head-> next;
             delete head;
             head=temp;
             }
       return Value;
```

```
      }
//------- показать значение первого элемента------------------------
      int peekHead (Node * head) {
      int Value=-1;
         if (head)       Value=head->data;
      return Value;
      }
//------- очистка очереди-----------------------------------------
      void clear (Node *&head, Node *&tail){
      Node* temp; tail=NULL;
        while (head) {
          temp=head-> next;
          delete head;
          head=temp;
         }
       }
//---------------------------------------------------------------
```

При программировании очереди необходимо предусмотреть обработку двух исключительных ситуаций:

- при добавлении элемента контролировать переполнение массива или динамической памяти;

- при удалении элемента контролировать наличие элементов в очереди.

3.2. Дек, двусторонняя очередь

Дек – это двусторонняя очередь, где операции включения и исключения элементов выполняются с обоих концов структуры.

Дек (Deque) функционирует одновременно по двум принцам организации данных - FIFO и LIFO. Определить дек можно как очередь с двумя сторонами, или стек, имеющий два конца. Дек (deque, double-ended queue) – это очередь с двумя концами, рис. 42.

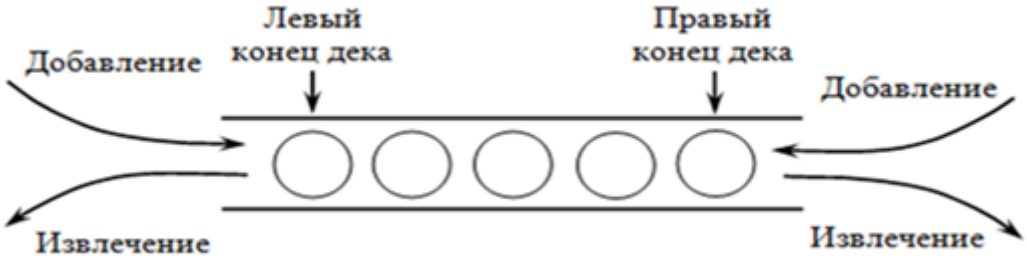


Рис.42 Графическая модель дека

Дек поддерживает следующий набор операций:

§ добавить элемент в начало (левый конец) дека - pushHead();

§ извлечь элемент из начала дека с возвратом значения - popHead();

§ добавить элемент в конец (правый) дека - pushTail();

§ извлечь элемент из конца дека с возвратом значения - popTail();

§ прочитать значение начала и конца дека (если не пуста) – peekHead(), peekTail();

§ проверить, пуст ли дек – isEmpty();

§ очистка дека – clear();

§ вывод размера дека - count(), length().

Дек реализуют на основе двух базовых структур: кольцевой массив и двусвязный список. Реализация дека - это алгоритмы реализации очереди и стека.

Модель дека на двусвязном списке представлена на рис. 43.

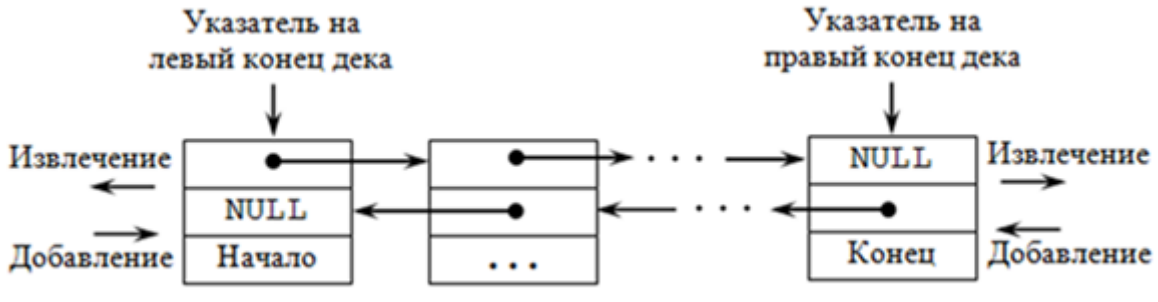


Рис.43 Модель дека на двусвязном динамическом списке

Простейшая реализация дека на кольцевом массиве как АТД представлена в Программе 2.7.

Программа 2.7 Реализация дека как АТД Deque на базовой структуре – кольцевой одномерный массив. Здесь функции операций реализованы в составе структуры. Обращение к полям и функциям структуры идет через объект.

```
//---------------------------------------------------------------------
struct Deque {
    int a[1000];
//размер позволяет иметь резервные элементы как слева, так и справа
    int head = 500;    //указатель первого элемента – индекс  массива
    int tail = 500;    //указатель элемента, следующего за последним
    void pushHead(int val) {
        head--;
        a[head] = val;
    }
    void pushTail(int val) {
        a[tail] = val;
        tail++;
    }
    int popHead() {
        if (head != tail) {
            head++;
```

```
            return a[head - 1];
        } else {
           //Исключительная ситуация:
           // попытка извлечь элемент из пустого дека
        }
    }
    int popTail() {
        if (head != tail) {
            tail--;
            return a[tail];
        } else {
           //Исключительная ситуация:
           // попытка извлечь элемент из пустого дека
        }
    }
    bool isEmpty() {
        return head == tail;
    }
       void clear() {
         head =500; tail =500;
    }
    int peekTail(){
          return a[tail-1];
    }
};
 int main(){
     Deque D;               //объявили объект АТД
     D.pushHead(25); D.pushHead(45);  //обращение к функциям объекта
     D.pushTail(35); D.pushTail(55);
     cout << D.peekTail()<<endl;
     while (!D.isEmpty()) cout << setw(5) << D.popHead();
 }
//---------------------------------------------------------------------
```

4. Бинарное дерево

Дерево – это иерархическая структура данных, где все элементы расположены по уровням и происходят из одного родительского элемента – корня дерева, он расположен на нулевом уровне. Каждый элемент нижнего уровня имеет ровно одного предка.

Основные определения

Элемент дерева называется *вершиной (узлом) дерева*, вершины дерева соединены направленными дугами - *ветвями дерева*. *Листья* дерева - вершины, в которые входит одна ветвь и не выходит ни одной ветви.

Свойства дерева:

- существует вершина, в которую не входит ни одной дуги (корень);
- в каждую вершину, кроме корня, входит одна дуга.

Вершины, в которые входят ветви, называются потомками, а сама вершина – предком. Уровень потомка на единицу больше уровня его предка. Корень дерева не имеет предка, а листья дерева не имеют потомков.

Количество уровней в дереве называют глубиной (высотой) дерева. Пустое дерево имеет глубину ноль, глубина дерева из одного корня – единица. Уровни нумеруют, начиная с 0, рис. 44.

Степенью (арностью) вершины в дереве называют количество дуг, выходящих из нее. Арность дерева определяется по максимальной арности (степени) вершины, входящей в дерево. В зависимости от степени вершины выделяют следующие разновидности деревьев:

-бинарные деревья – степень дерева не более двух, рис.45 (б);

-N-арные, произвольные деревья – степень дерева N, рис. 45 (а).

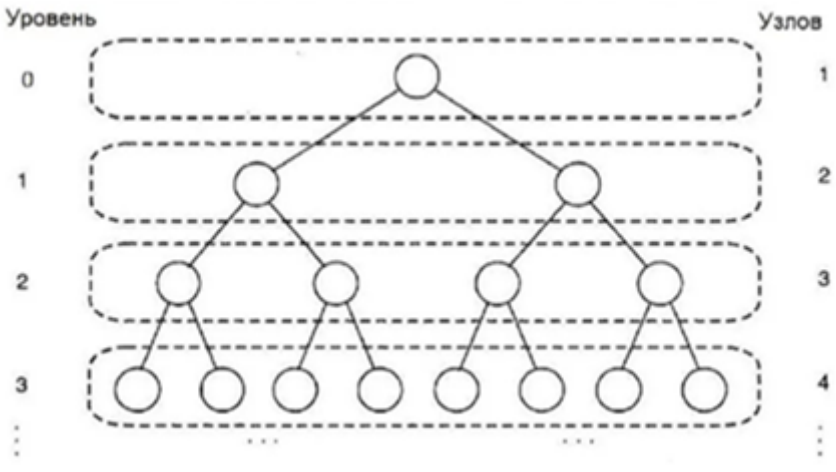


Рис.44 Бинарное дерево

Зная арность дерева (N) и номер уровня (*l*) можно рассчитать *максимальное число вершин* (узлов) в дереве на этом уровне по формуле:

Например, в бинарном дереве на 3-м уровне будете не более 8 вершин (

Поддерево – часть древообразной структуры данных, которая представлена в виде отдельного дерева. Следовательно, деревья - это рекурсивные структуры, где каждое поддерево является деревом. Операции с рекурсивными структурами реализуют рекурсивными алгоритмами. Если дерево определить как рекурсивную структуру, то каждый элемент является:

1) либо пустым деревом;
2) либо элементом, с которым связано конечное число поддеревьев.

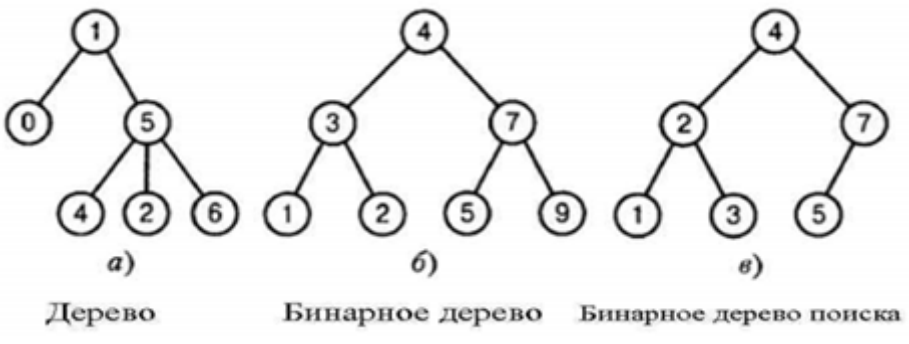


Рис.45 Разновидности деревьев

Операция обхода дерева – это упорядоченный доступ к каждой вершине дерева. Доступ к вершинам дерева осуществляется в определенном порядке. Существует три порядка обхода дерева (рис. 46):

- прямой (preorder) – порядок обхода, при котором корень предшествует обоим поддеревьям;
- обратный (postorder) – порядок обхода, при котором корень следует за поддеревьями;
- симметричный (inorder) - порядок обхода, при котором сначала следует самая левая вершина, корень, правая вершина.

Все операции обхода являются рекурсивными алгоритмами.

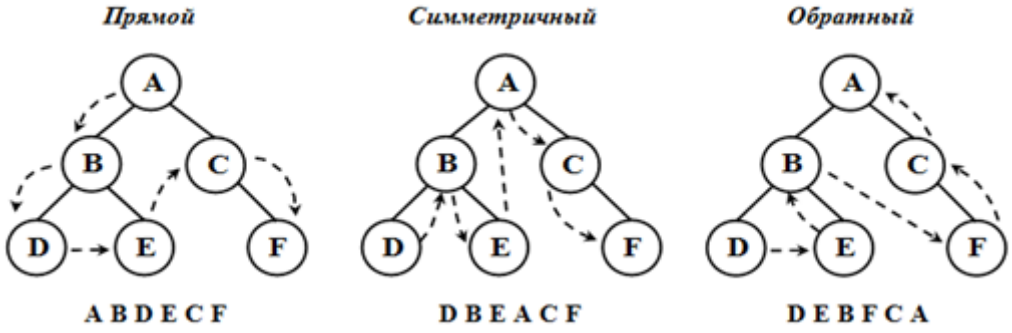


Рис. 46 Обходы деревьев

Для алгоритмизации используют простейшую разновидность дерева – бинарное. Если модель данных не соответствует бинарному дереву, то его определенными методами преобразуют из n-арного к бинарному дереву для обработки.

Бинарное (двоичное) дерево — иерархическая структура данных, в которой каждая вершина (узел, Node) имеет не более двух потомков (детей). Корень (root) дерева/поддерева называется родительским узлом, а потомки называются левым (left) и правым (right) наследниками (потомками), рис. 45 (б).

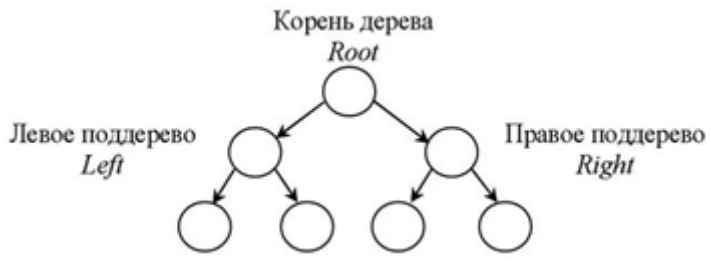


Рис. 47 Обозначения в бинарном дереве

Для обработки бинарные деревья различают, рис. 48:

§ Полное и неполное бинарное дерево. Полное дерево содержит только полностью заполненные уровни, иначе оно является неполным.

§ Строгое и нестрогое бинарное дерево. Строгое дерево имеет вершины степенью 0 (листья) или 2; нестрогое имеет вершины со степенью 0, 1 или 2.

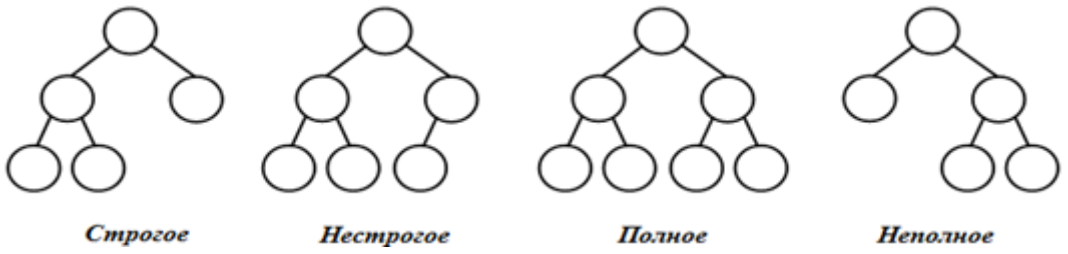


Рис. 48 Разновидности бинарных деревьев

Основные операции с бинарными деревьями:

§ создание бинарного дерева на базовой структуре данных – makeTree();

§ обход бинарного дерева (прямой, симметричный, обратный) – preorderTree (), inorderTree (), postorderTree ();

§ вставка элемента в бинарное дерево – insertTree();

§ удаление элемента из бинарного дерева – removeTree();

§ проверка пустоты бинарного дерева – isEmpty();

§ удаление бинарного дерева – clear().

Бинарные деревья программируют на двух базовых структурах: массивах и связных списках. Связные списки для организации дерева нелинейные двусвязные или трехсвязные.

Реализация бинарного дерева на массиве

Лемма. Двоичное дерево массива длины M имеет глубину (число уровней) $\log_2 M$.

Используя эту зависимость, можно определить размер массива для хранения бинарного дерева.

Алгоритм *операции создания* makeTree() - заполнения массива значениями вершин - бинарного дерева:

1. В качестве поля структуры Tree для хранения дерева выступает одномерный массив a[] фиксированной длины. Заполняем его значениями вершин бинарного дерево по правилу представленному на рис. 49.

2. Корень дерева хранится в элементе массива a[0]. Индекс корня i = 0.

3. Индекс левого потомка рассчитывается по формуле 2*i+1, где i – индекс корня текущего поддерева.

4. Индекс правого потомка рассчитывается по формуле 2*i+2, где i – индекс корня текущего поддерева.

5. Доступ к родительской вершине из текущей по правилу (i-1) div 2, где i – номер текущей вершины.

Описываем *операции дерева в виде функций обработки одномерного массива* a[] или АТД Tree. Удобно программировать доступ к левой, правой и родительской вершине через набор функций, представленный в Программе 2.15.

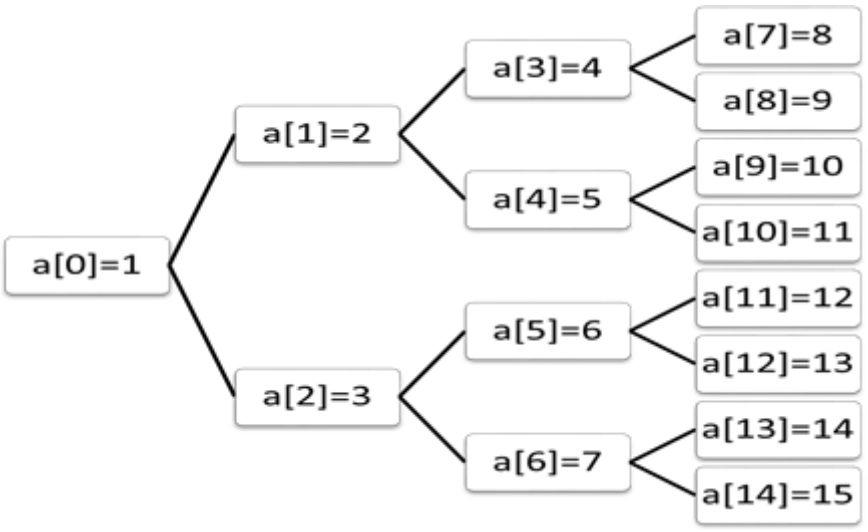


Рис. 49 Соответствие индексов массива и вершин бинарного дерева

Рассмотрим в качестве примера еще одно дерево на рис. 50. В массиве будут следующие значения int A[] ={5, 1, 3, 9, 6, 2, 4, 7, 0, 8};

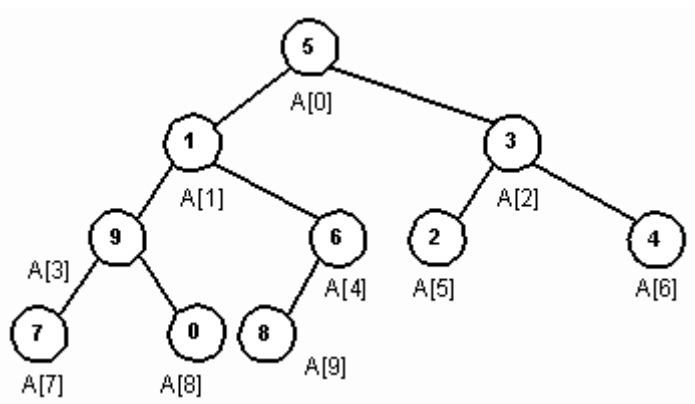


Рис. 50 Бинарное дерево на массиве

В случае неполного дерева в массиве будут пустые значения, поэтому нужно обязательно ввести обозначение признака пустого значения и по нему определять незначащие элементы массива. К недостаткам реализации дерева в массиве относятся:

- неполное дерево неэффективно использует память массива, много пустых элементов;
- размер дерева ограничен размером массива.

Реализация бинарного дерева на динамическом связном списке

Списочное представление бинарного дерева основано на описании узлов (Node) списка, соответствующих вершинам дерева, и указателя корня дерева root на начало списка. Узлы листовых вершин содержат в указателях левой и правой вершины NULL. Возможно два варианта описания узлов такого списка. Второй вариант описания структуры используют для работы с деревьями поиска.

Описание узлов и корня дерева:

1. Каждый узел имеет поле данных и два поля указателей (рис. 51): на левую вершину и правую.

```
struct Node {
  int d;
  Node* left;       //указатель на правого потомка
```

```
  Node* right;      //указатель на левого потомка
  };
Node* root=NULL;   //указатель на дерево
```

2. Каждый узел имеет поле данных и три поля указателей (рис. 52): на левую и правую вершину, а также указатель на родительскую вершину. Корень дерева в указателе на родителя содержит NULL.

```
struct Node {
  int d;
  Node* father;    //указатель на родителя
  Node* left;
  Node* right;
  };
Node* root=NULL;
```

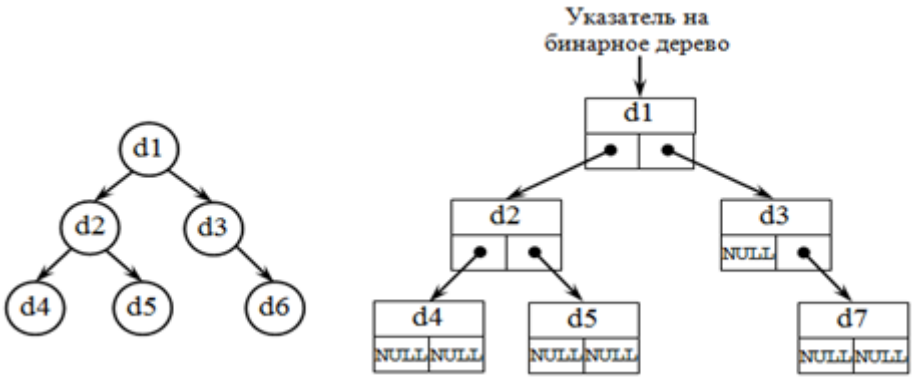


Рис. 51 Бинарное дерево на связном списке – 2 указателя

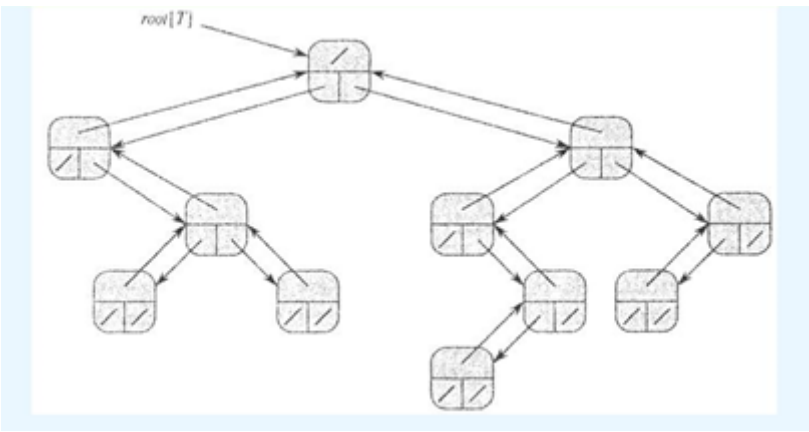


Рис. 52 Бинарное дерево поиска на связном списке – 3 указателя

Все операции дерева программируют на основе алгоритмов обработки связных списков.

Из одного и того же набора узлов можно сконструировать различные варианты двоичных деревьев. В простейшем случае дерево можно построить так, что его глубина будет равна числу элементов, и дерево выродится в односвязный список. На практике добавление и удаление элементов в дерево может происходить в произвольном порядке, что приводит к созданию различных деревьев, в том числе и вырожденных.

Операция создания бинарного дерева ставит задачу разработать полное дерево с N вершинами (узлами) и минимальной глубиной. Минимальная глубина позволит осуществить максимально быстрый доступ к каждой вершине. Это создание *сбалансированного бинарного дерева*.

Сбалансированное дерево – это дерево с максимально возможным числом вершин на каждом уровне, кроме нижнего уровня. В таком дереве для каждой вершины в левом и правом поддереве число вершин различается не более чем на 1.

Рекурсивный *алгоритм построения сбалансированного бинарного дерева* с N вершинами:

1. Пусть одна вершина будет корнем поддерева.
2. Создаем левое поддерево с NL=N/2 вершинами этим алгоритмом.
3. Создаем правое поддерево с NR=N-NL-1 вершинами этим алгоритмом.

Программа 2.8 Функция построения сбалансированного бинарного дерева с N вершинами на связном списке.

```
//-----------------------------------------------------------------
```

```
Node* makeTree( Node* root, int N){
 int NL, NR, val;
 if (N==0) return NULL;
  NL=N/2;
  NR=N-NL-1;
  Node* temp= new Node;
  cout<<”Значение вершины:”<<endl; cin>>val;
  temp->d=val;
  temp->left=makeTree(root, NL );
  temp->right=makeTree(root, NR);
 return temp;
}
//---------------------------------------------------------------
…
// первый вызов функции
 root = makeTree(root, 12);
//---------------------------------------------------------------
```

Псевдокод рекурсивной *операции прямого обхода* бинарного дерева, где p - указатель на корень поддерева:

```
void preorderTree (Node * p) {
  if (p)   {
       // действия с узлом вершины
       preorderTree (p->left);
       preorderTree (p->right);
    }
 }
```

Псевдокод рекурсивной *операции обратного обхода* бинарного дерева, где p - указатель на корень поддерева:

```
void postorderTree (Node * p) {
   if (p)  {
       postorderTree (p->left);
       postorderTree (p->right);
       // действия с узлом вершины
   }
 }
```

Псевдокод рекурсивной *операции симметричного обхода* бинарного дерева, где p - указатель на корень поддерева:

```
void inorderTree (Node * p) {
       if (p)  {
       inorderTree (p->left);
```

```
    // действия с узлом вершины
    inorderTree (p->right);
  }
}
```

Рассмотрим функцию *операции прямого обхода* бинарного дерева для организации вывода на экран.

Программа 2.9 Функция вывода бинарного дерева на основе прямого обхода, где p – указатель корня поддерева, l – номер уровня. Вывод на экран дерева будет горизонтальный, количество отступов от начала строки – номер уровня l.

```
//---------------------------------------------------------------------
void preorderTree(Node * p, int l) {
    if (p) {
        for (int i = 0; i < l; i++) cout << ' '; //вывод отступов
        cout << p->d << endl;         //вывод значения из вершины
    // рекурсивный вызов увеличивает номер уровня
        preorderTree(p->left, l + 1);
        preorderTree(p->right, l + 1);
    }
}
int main() {
  root = makeTree(root, 5);
  preorderTree(root, 0);
}
//---------------------------------------------------------------------
```

Для программирования произвольного дерева его преобразуют в бинарное дерево. Один из методов преобразования – использование представления дерева моделью “left child – right sibling representation” (LCRS, «левый ребенок – правый сосед»), рис. 53.

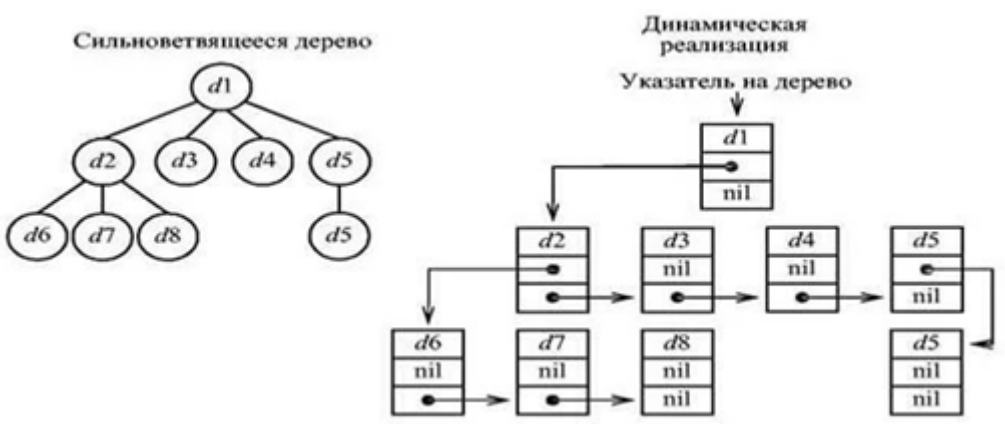


Рис. 53 Модель дерева “left child – right sibling representation”

Алгоритм преобразования дерева с произвольным ветвлением к бинарному дереву:

1. левый ребенок каждой вершины остается тем же;

2. правым ребенком текущей вершины становится её правый сосед (т.е. элемент, являющийся следующим ребенком одного и того же родителя).

На выходе будет дерево, где каждая вершина имеет не более двух потомков.

В программирование наибольшее применение имеют два вида двоичных деревьев: сбалансированное дерево поиска и сортирующее дерево (пирамида).

4.1. Бинарное дерево поиска

В основе представления дерева поиска лежат свойства упорядоченности и сбалансированности дерева. Назначение таких деревьев - эффективный поиск.

Свойство сбалансированности позволяет минимизировать время поиска и других операций в таком дереве. Время поиска по двоичному дереву зависит от глубины (высоты) дерева, и чем глубже (выше) дерево - тем больше времени требуется на поиск определенного узла в дереве и наоборот.

Упорядоченное дерево – это дерево, у которого ветви, исходящие из каждой вершины, упорядочены по определенному критерию.

Бинарное дерево поиска, BST - дерево (Binary Search Tree) – это двоичное дерево (рис. 54**)**, где:

a. Каждая вершина/узел (Node) имеет не более двух потомков (child nodes);

b. Каждая вершина/узел имеет ключ (key) и значение (value), где ключ соответствует критериям:

- ключи всех узлов левого поддерева меньше значения ключа родительского узла;
- ключи всех узлов правого поддерева больше или равны значению ключа родительского узла.

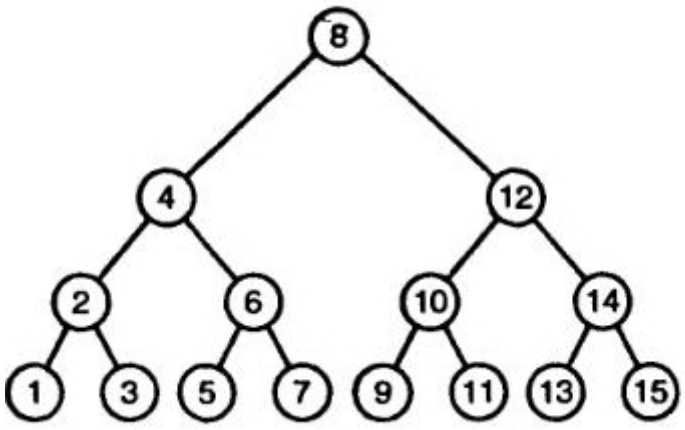


Рис. 54 Сбалансированное дерево поиска

В простейшем случае ключом будет значение вершины в узле.

Пусть **x** – произвольная вершина двоичного дерева поиска:

- если вершина ***y*** находится в левом поддереве вершины ***x***, то ***y*** < ***x****:*

$$y->d < x->d$$

- если в правом, то

$$y->d >= x->d.$$

Знак равенства можно ставить в любом условии.

Операции с бинарными деревьями поиска:

§ добавление/вставка нового элемента - insertBST();

§ обход бинарного дерева поиска (прямой, симметричный, обратный): preorderTree(), inorderTree(), postorderTree();

§ поиск элемента в дереве – searchBST();

§ поиск максимального, минимального элемента – minBST(), maxBST();

§ поиск следующего и предыдущего по значению – succBST(), predBST();

§ удаление элемента – deleteBST();

§ удаление всего дерева – clear();

§ проверка на пустоту – isEmpty().

Реализация этих операций может быть итеративной – циклом, и рекурсивной. На некоторых операция будут показаны оба варианта реализации.

Описание узла дерева поиска:

```
struct Node {
    int d;
    Node* father;
    Node* left;
    Node* right;
    Node() { father = right = left = NULL; }
};
Node* root = NULL;
```

Операция добавления/вставки элемента insertBST() находит для значения/ключа нового элемента подходящее место и вставляет его, сохраняя свойство упорядоченности.

Программа 2.10 Функция вставки нового значения в дерево поиска.

```
//---------------------------------------------------------------------
void insertBST(Node*& root, int val) {
    Node* z = new Node;    z->d = val;
    if (root == NULL) { root = z; return; }    //пустое дерево
    Node* y = NULL, * x = root;
    while (x) {           //поиска корня  для вставки
        y = x;
        if (z->d < x->d) x = x->left;
        else x = x->right;
    }
    z->father = y;          // присвоили указатель на родителя
    if (y->d > z->d)       // подцепляем к  родителю слева или справа
        y->left = z;
    else y->right = z;
}
//---------------------------------------------------------------
```

Операции обхода дерева поиска повторяют операции простого бинарного дерева. Но свойство упорядоченности позволяет обрабатывать все значения в неубывающем порядке в процессе симметричного обхода дерева. В этом случае мы обходим отсортированный список значений узлов.

Операция поиска searchBST() – основная целевая функция дерева поиска, рис.55 . Эту операцию рассмотрим в рекурсивной и итеративной реализации. Итеративная версия традиционно более быстрая.

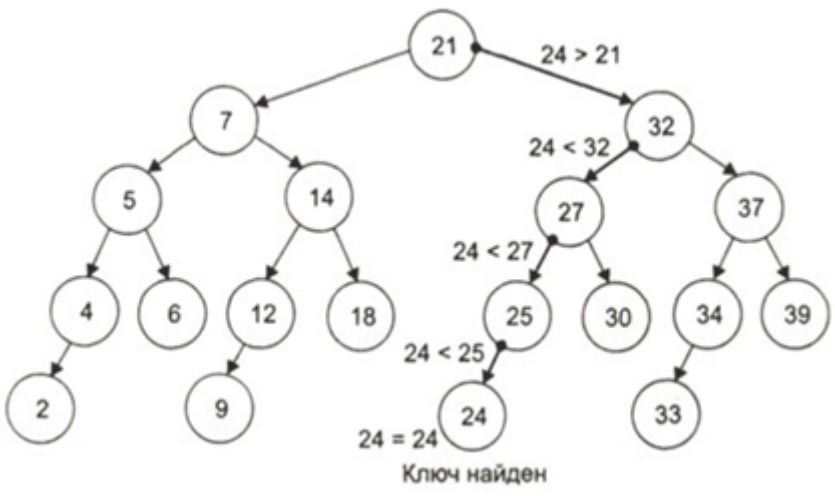


Рис. 55 Модель поиска значения 24 в BST - дереве

Программа 2.11 Функции поиска значения в дереве поиска, реализованные рекурсивно и итеративно, где p – корень дерева или корень поддерева, key- искомое значение .

```
//---------------------------------------------------------------------
Node* searchBST(Node* p, int key) {
    if (p == 0 || p->d == key) return p;
    if (key < p->d) return searchBST(p->left, key);
    else return searchBST(p->right, key);
}
//--------Итеративная версия (более эффективна – быстрая)-----------
Node* iterative_searchBST(Node* p, int key) {
    while (p != 0 && key != p->d)
        if (key < p->d) p = p->left;
        else p = p->right;
    return p;
}
//---------------------------------------------------------------------
```

Операции поиска максимального и минимального по значению элемента в дереве являются симметричными по своей реализации. Минимальное значение в дереве поиска можно найти, пройдя всю глубину дерева по указателям left до листьев. Максимальное значение найдем аналогично, но спускаться надо по указателям right.

Программа 2.12 Функция поиска минимального элемента в дереве поиска.

```
//---------------------------------------------------------------
Node* minBST(Node* p) {
    while (p->left != 0)
        p = p->left;
    return p;
}
//---------------------------------------------------------------
```

Операция поиска следующего по значению за вершиной *p* требуют учесть в алгоритме два случая:

- если правое поддерево *p* – не пусто, то следующий за *p* элемент – минимальный в этом поддереве.
- если правое поддерево – пусто, тогда мы идём вверх от *p*, пока не найдём вершину, являющуюся правым сыном.

Операция поиска предыдущего – симметрична операции поиска следующего.

Программа 2.13 Функция поиска следующего значения относительно вершины *p* в дереве поиска.

```
//-----------------------------------------------------------
Node* succBST(Node* p) {
    if (p->right) return minBST(p->right);
    Node* y = p->father;
    while (y != 0 && p == y->right) {
        p = y;
        y = y->father;
    }
    return y;
}
//-----------------------------------------------------------
```

Алгоритм *операции удаление элемента* должен предусмотреть три случая расположения удаляемых элементов, см. рис. 56:

1. *p1* – листовая (терминальная) вершина, ее удаляют, обнуляя указатель у ее предка;
2. *p2* – корень неполного поддерева (один потомок), перенаправляют указатели предка, вырезая удаляемую вершину и удаляют её;
3. *p3* – корень большого поддерева, идет поиск следующей по значению вершины, найденное значение переносят в корень поддерева, а ее удаляют.

Программа 2.14 Функция удаления элемента из дерева поиска, p – указатель на удаляемый элемент, y- фактически удаляемый элемент, x – указатель потомка удаляемого элемента.

```
//---------------------------------------------------------------
void deleteBST(Node*& root, Node* p) {
    Node* x = NULL, * y = NULL;
    if (root && p) {
        if (p->left == 0 || p->right == 0)   //фактически  удаляемая
            y = p;
        else      y = succBST(p);
        if (y->left) x = y->left;            //находим потомка
        else   x = y->right;
        if (x)   x->father = y->father;      //вырезаем вершину
        if (y->father == 0)    root = x;     // если корень
        else {
            if (y == y->father->left)      // перенаправляем связи
                y->father->left = x;         // на потомка
            else y->father->right = x;
            if (y != p) p->d = y->d;         //переносим данные
            delete y;
        }
    }
}
//---------------------------------------------------------------
```

В алгоритме удаления сначала находим удаляемую вершину *y* – это либо явно вершина на входе, либо вершина, следующая по значению (succBST). Далее находим указатель на существующего потомка (*x)* вершины *y.* Вырезаем вершину *y,* меняя указатели. Рассматривается случай, когда *y* – корень дерева. Если случай *p3*, переносим данные.

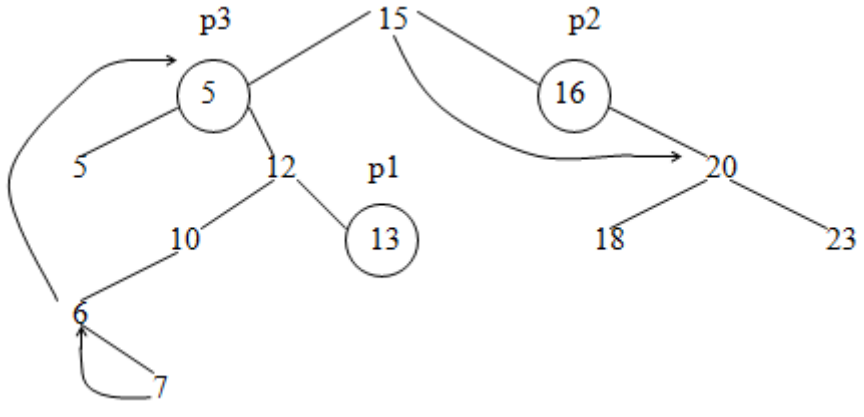


Рис. 56 Случаи удаления элементов в дереве поиска

Алгоритм *удаления всего дерева* использует обратный обход, так как при обратном обходе сначала посещаются потомки, а потом родительский узел. Это дает возможность удалить сначала нижние по уровню вершины, а затем корень. При этом для удаления очередной вершины используем операцию *delete.*

Дерево необходимо балансировать в процессе операций добавления/удаления, но требование идеальной сбалансированности делает этот процесс достаточно сложным, так как должны быть обработаны все вершины дерева.

Два основных типа сбалансированного дерева поиска, используемых в программировании: AVL-деревья[1] и красно-черные деревья.

AVL-дерево (АВЛ-дерево) - сбалансированное бинарное дерево поиска, где для каждого узла высота его двух поддеревьев различается не более чем на 1. Идеально сбалансированное бинарное дерево с N узлами имеет глубину (высоту) равную целому числу log(N+1) по основанию 2.

Каждый узел AVL-дерева определяется *показателем сбалансированности* – это разность между глубиной (высотой) правого и левого поддерева данного узла. Пусть hR - глубина правого поддерева, hL - глубина левого поддерева. Дерево будет AVL-сбалансировано, если показатель каждого узла соответствует неравенству:

$$|hR - hL| <= 1.$$

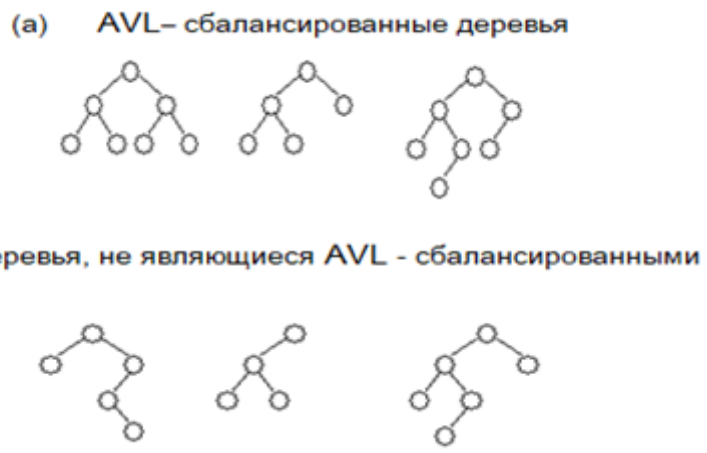


Рис. 57 Примеры AVL-деревьев и других

Показатель сбалансированности (hR – hL) в AVL-дереве может принимать одно из трех значений:

· -1, если левое поддерево на единицу выше правого;

· 0, если высоты обоих поддеревьев одинаковы;

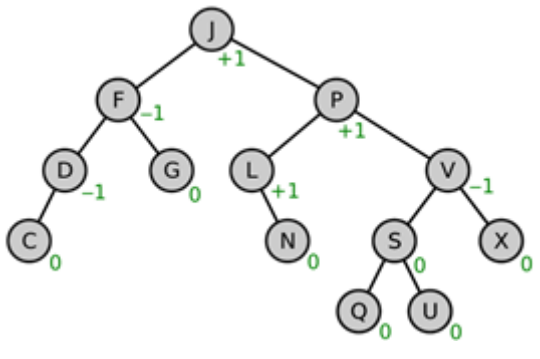
· +1, если правое поддерево на единицу выше левого.

Если хотя бы один узел имеет другое значение показателя, то дерево не является AVL-сбалансированным. Примеры AVL-сбалансированных и AVL-несбалансированных деревьев представлены на рис. 58.

У листовых узлов показатель всегда равен 0. У остальных узлов вычисляется по формуле (hR – hL). Показатель сбалансированности (balance) будет полем в описание структуры узла. Указатели на левую и правую вершину представлены в виде массива указателей. Если необходимо, можно добавить в массив еще один указатель на родительский узел. Все операции добавления/удаления узлов сопровождаются пересчетом поля показателя сбалансированности.

Описание узла AVL-дерева:

```
struct NodeAVL {
    int data;
    NodeAVL* link[2];     //массив указателей на левое и правое поддерево
    int balance;          //показатель сбалансированности
};
NodeAVL* root = NULL;
```



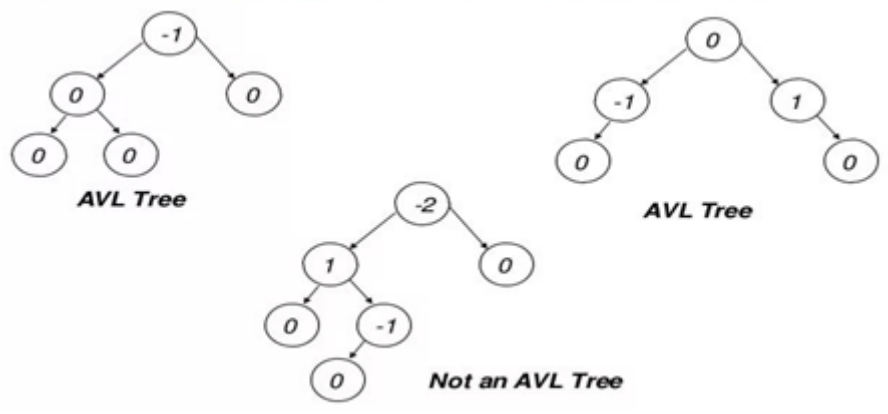


Рис. 58 Примеры AVL-сбалансированных и AVL-несбалансированных деревьев

Алгоритм добавления нового узла в AVL-дерево:

1. поиск позиции в дереве, где должен находиться новый узел;
2. добавить узел на найденную позицию;
3. пересчитать показатели сбалансированности для узлов, находящихся выше добавленного узла;
4. если дерево стало несбалансированным, то выполнить балансировку.

Балансировка выполняется алгоритмами, называемых поворотами узлов. Описание этих алгоритмов вне данного учебного пособия.

Красно-черное дерево (Red-Black-Tree, RB-Tree) – это бинарное дерево со следующими свойствами, рис.59:

a. каждая вершина должна быть окрашена либо в черный, либо в красный цвет;

b. корень дерева должен быть черным;

c. листья дерева должны быть черными и объявляться как NULL-вершины (наследники узлов, которые обычно называют листьями; на них "указывают" NULL указатели);

d. каждый красный узел должен иметь черного предка;

e. на всех ветвях дерева, ведущих от его корня к листьям, число черных вершин одинаково.

Характеристика красно-черных деревьев:

· Красно-черные деревья являются методом балансировки деревьев, что определяется свойствами данной структуры.

· Красно-черные деревья обрабатываются теми же операциями, что и бинарные деревья.

· Операции добавления/удаления элемента требуют балансировки дерева.

Описание узла RB-дерева:

```
struct NodeRB {
    int data;
    NodeRB* link[2];    //массив указателей на левое и правое поддерево
    char red;          //показатель сбалансированности
};
NodeRB* root = NULL;
```

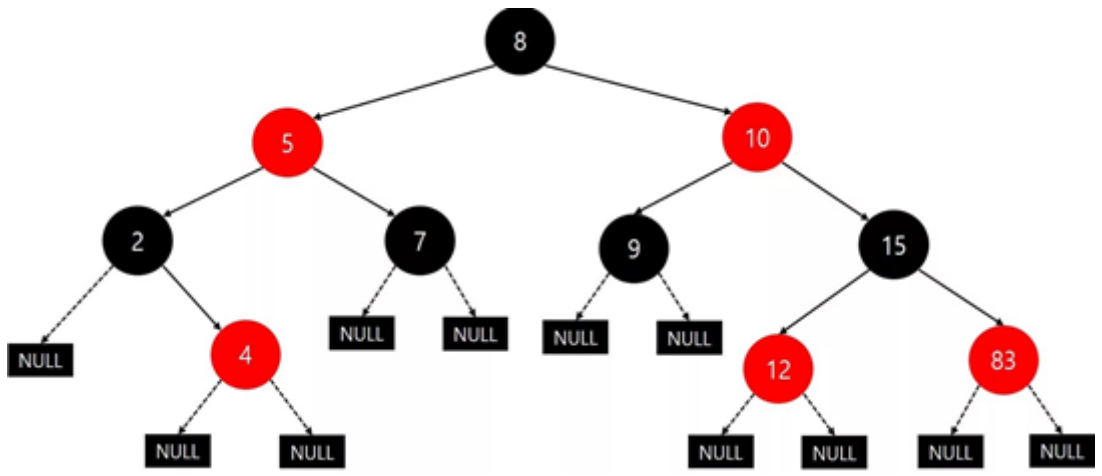


Рис. 59 Красно-черное дерево

Сравнительная характеристика методов балансировки AVL-дерева и красно-черных деревьев:

- эффективность алгоритмов вставки/удаления одинаковая, вычисляется в как двоичный логарифм от общего числа вершин в дереве;

- если число вершин дерева одинаково, максимальная глубина AVL- дерева всегда будет меньше, чем максимальная глубина красно-черного дерева, поэтому для поиска по красно-черному дереву требуется большее число операций, что сказывается на быстродействии алгоритмов;

- если число вершин дерева одинаково, хранение AVL-дерева требует больше памяти, так как каждый узел должен хранить глубину как целое число, а в узле красно-черного дерева хранится только один байт (можно ограничиться битом!), определяющий цвет узла.

---

[1] AVL, АВЛ – сокращение из первых букв фамилий создателей структуры - советских математиков Г.М. Адельсона-Вельского и Е.М. Ландиса.

4.2. Сортирующее дерево, пирамида

Пирамида, двоичная куча (binary heap) - это упорядоченное двоичное дерево, у которого минимальный/максимальный элемент расположен в корне. Для дерева обязательны три свойства:

a. каждая вершина по значению не больше/не меньше, чем значения ее потомков.

b. разница между глубиной листьев не более 1 уровня.

c. нижний уровень формируется слева направо, стремясь к созданию полного дерева.

Пирамиду, у которой в корне расположен минимальный элемент, называют минимальной или неубывающая (min-heap), а где в корне максимальный элемент – максимальной или невозрастающая (max-heap) (рис.60).

Основная цель построения такого дерева – упорядочить, отсортировать исходную неупорядоченную последовательность данных путем создания пирамиды. Алгоритмы обработки пирамиды должны сами обновлять дерево и поддерживать упорядоченность вершин. Реализуют алгоритмы пирамиды на базе одномерного массива.

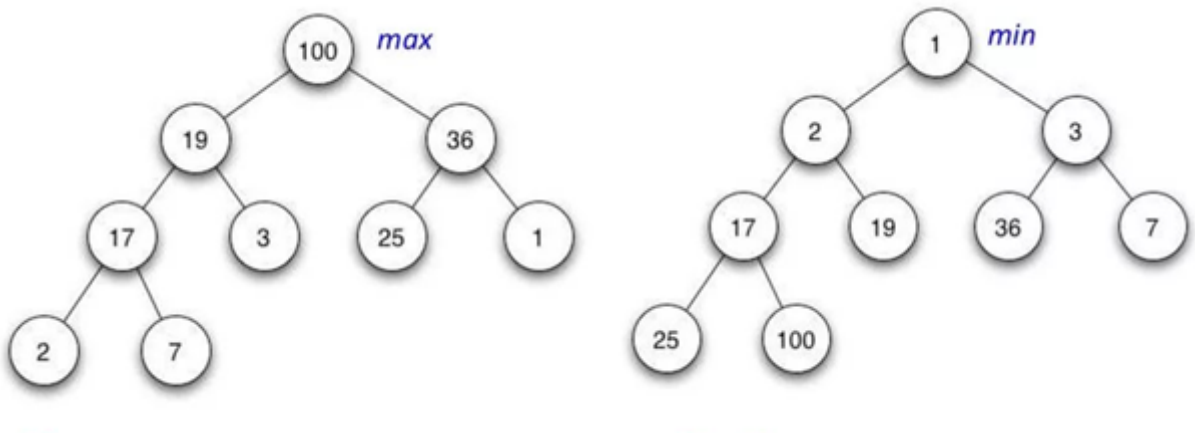


Рис. 60 Максимальная и минимальная пирамида

Основные операции с пирамидой:

§ построить пирамиду – buildHeap(), преобразовать неупорядоченный массив в пирамиду;

§ добавить элемент в пирамиду – pushHeap();

§ удалить корневой (минимальный/максимальный) элемент – popHeap();

§ упорядочивание пирамиды –heapify();

§ показать корневой элемент – peekHeap();

§ пирамидальная сортировка – sortHeap().

Алгоритм построения хранит пирамиду в одномерном массиве по следующим правилам:

1) корень имеет индекс 0;

2) потомки (left, right) i-го корня поддерева имеют индексы 2*i+1, 2*i+2;

3) родительская вершина (father) i-го элемента (i-1)/2.

Пример представлен на рис. 61.

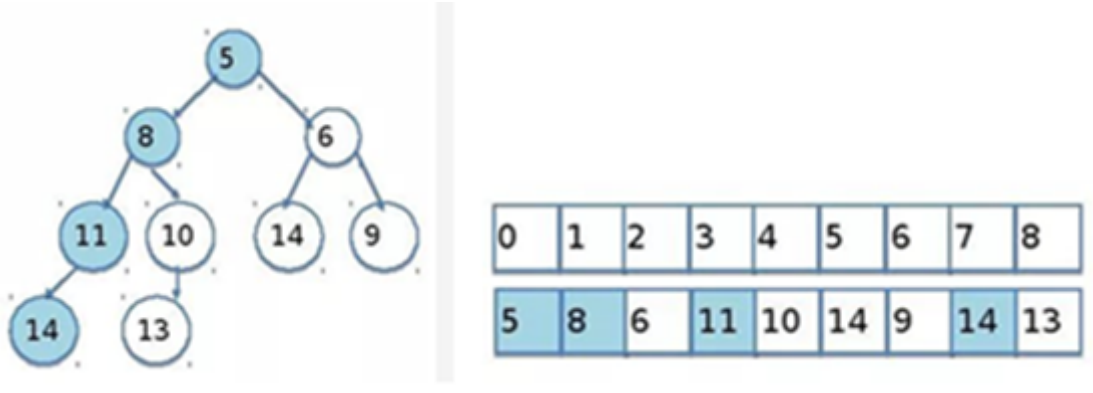


Рис. 61 Представление пирамиды в массиве

Программа 2.15 Функции расчета индексов связанных вершин пирамиды в массиве по индексу i текущей вершины.

```
//-----------------------------------------------------
int father(int i) {
    return (i - 1) / 2;
}
int left(int i) {
    return 2 * i + 1;
}
int right(int i) {
    return 2 * i + 2;
}
//-----------------------------------------------------
```

Алгоритм преобразования buildHeap() исходного неупорядоченного массива размером N в пирамиду основан на перестановке элементов массива по *правилам алгоритма построения* минимальной/максимальной пирамиды. Последний элемент пирамиды – лист - имеет индекс (N-1), следовательно, его предок расположен по индексу (N-2)/2. Рассмотрим данный алгоритм на примере.

Пусть имеем массив int H[10]={9, 12, 17, 30, 50, 20, 60, 65, 4, 19}; Пирамида будет иметь три уровня (log N) – 0, 1, 2. Преобразуем этот массив в минимальную пирамиду. Рассчитаем индексы листовых вершин (5, 6, 7, 8, 9). Индексы корневых вершин соответственно (4, 3, 2, 1, 0).

1. Уровень 2: корень поддерева H[4]=50 > H[9]=19, поменяем их местами;

2. Уровень 2: корень поддерева H[3]=30 > H[8]=4, поменяем их местами (если потомка два {4, 65}, меняем местами с наименьшим из них);

Результат: H[10]={9, 12, 17, 4, 19, 20, 60, 65, 30, 50};

3. Уровень 1: Корень поддерева H[2]=17 и он меньше чем H[5]=20 и H[6]=60, все оставим на месте;

4. Уровень 1: Корень поддерева H[1]=12 > H[3]=4, поменяем их местами (здесь два потомка {4,19});

Результат: H[10]={9, 4, 17, 12, 19, 20, 60, 65, 30, 50};

5. Уровень 0: Корень пирамиды H[0]=9>H[1]=4, поменяем их местами. Процесс закончился, так как достигли корня пирамиды.

Результат: H[10]={4, 9, 17, 12, 19, 20, 60, 65, 30, 50};

Алгоритм добавления элемента в минимальную/максимальную пирамиду, рис 62:

1. Элемент добавляется в конец массива как лист пирамиды.

2. Сравнивается с родительской вершиной, если он меньше/больше, то меняется местами с ней.

3. В свою очередь измененный элемент сравнивается со своим родителем на условие пирамидальности, в случае необходимости меняется местами. Процесс завершается, если обмена (swap) не было.

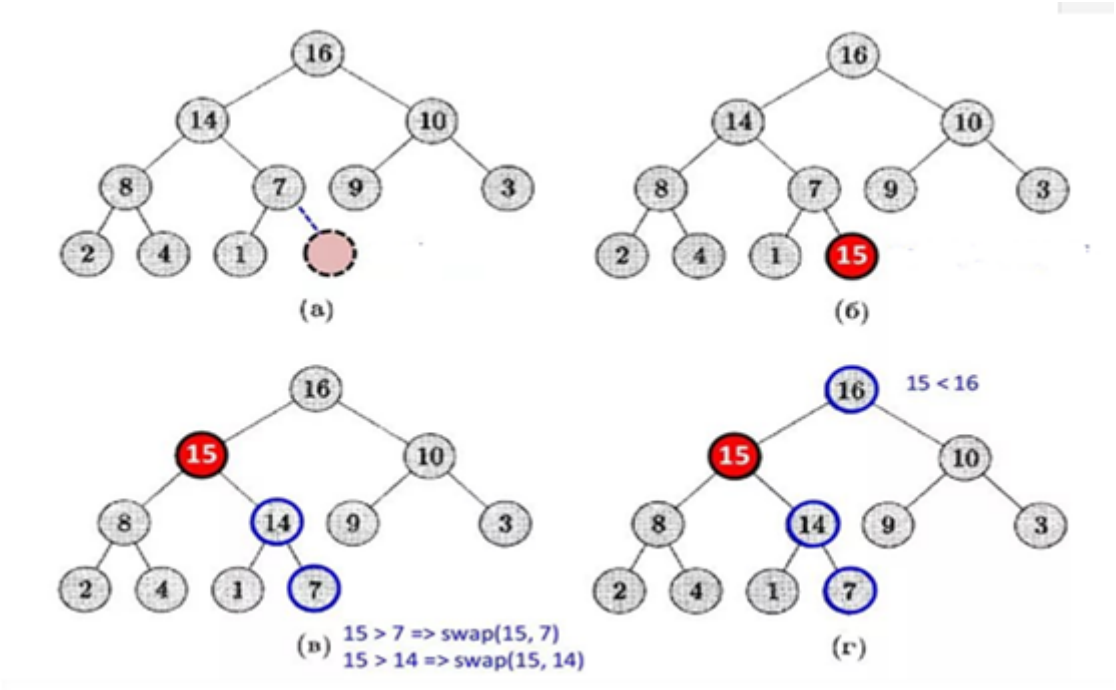


Рис. 62 Добавление элемента в пирамиду.

Алгоритм добавления можно представить как обратный проход по пирамиде снизу вверх – от листа до корня с целью поиска места нового элемента.

Алгоритм удаления элемента из корня минимальной пирамиды – это прямой проход по пирамиде от корня до листа для поиска нового корня пирамиды.

1. Удалить корневой элемент пирамиды и заменить его последним листом.

2. Если новый корень больше своих потомков, меняем местами с наименьшим потомком

3. Продолжаем сравнения с элементами потомков, до первого невыполнения условия и отсутствия обмена.

Алгоритм упорядочивание является базовым для всех алгоритмов пирамиды, так как его задача осуществить поиск места для i-го элемента в пирамиде. Имеет рекурсивную реализацию.

Программа 2.16 Функция упорядочивания пирамиды, где H – массив пирамиды, N – размер массива, i – индекс вершины, для которой идет поиск места в пирамиде.

```
//-----------------------------------------------------------------
void heapify(int H[], int i, int N) {
    int  temp;
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    if (left < N) {
        if (H[i] < H[left]) {
            temp = H[i];
            H[i] = H[left];
            H[left] = temp;
            heapify(H, left, N);
        }
    }
    if (right < N) {
        if (H[i] < H[right]) {
            temp = H[i];
            H[i] = H[right];
            H[right] = temp;
            heapify(H, right, N);
        }
    }
}
//-----------------------------------------------------------------
```

Алгоритм пирамидальной сортировки – это прямой обход пирамиды с последовательным удалением элементов.

1. Преобразовать массив в пирамиду - buildHeap();

2. Последовательно удалять корневой элемент пирамиды и упорядочивать пирамиду.

Порядок сортировке задает вид пирамиды минимальный - сортировка по возрастанию или максимальный – сортировка по убыванию.

На базе пирамиды строят еще одну абстрактную структуру данных – очередь приоритетов.

Очередь приоритетов - очередь, в которой все элементы имеют приоритет (key= Priority), где две обязательные операции — добавить элемент и извлечь в соответствии с максимумом (минимумом) приоритета.

Основные операции очереди приоритетов:

§ добавить элемент в очередь – Enqueue(key, value);

§ удалить из очереди элемент с минимальным /максимальным ключом –DeleteMin(), DeleteMax();

§ показать элемент с минимальным/максимальным ключом –Min(), Max();

§ изменить значение ключа элемента и упорядочить очередь –DecreaseKey().

Используя алгоритмы пирамиды, легко реализовать операции очереди приоритетов.