Информатика. 2-й курс.
Краткий конспект лекций.

Лектор - доц. Беляев Сергей Юрьевич

1 Оглавление

Рекомендуемая литература

Основная

1. Кормен, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн, Клиффорд. Алгоритмы: построение и анализ, 2-е издание. : Пер. с англ. — М. : Издательский дом "Вильяме", 2005. — 1296 с.
2. Н. Вирт. Алгоритмы и структуры данных. М.: Мир, 1989 г., с. 360.

Дополнительная

1. А. Ахо, Дж. Хопкрофт, Дж. Ульман. Построение и анализ эффективных алгоритмов. М.: Мир, 1979 г., с. 535.
2. Д. Кнут. Искусство программирования для ЭВМ. Т1. Основные алгоритмы. М.: Мир, 1976 г., с.734.

Введение

Временная и емкостная сложность

1. Списки

1.1 Линейные списки, стеки и очереди

1.1.1 Сплошное и связное представления

- **Добавить q после p**
  q->next=p->next;
  p->next=q;
- **Добавить q перед p**
  q->next=p->next;
  p->next=q;
  q->data<->p->data
- **Удалить после p**
  q=p->next;
  p->next=q->next;
  free(q);
- **Двусвязные и кольцевые списки**

- **Списки на массивах**

1.1.2 Добавление с поиском

```
struct List{
        int data;
        List *next;
};
List* Add(List* p, int x)
{
        List *q=p;
        while (q!=NULL)
        {
                if (q->data==x)
                        break;
                else
                        q=q->next;
        }
        if (q==NULL)
                q=(List*)malloc(sizeof(List));
                q->data=x;
                q->next=p;
                p=q;
        }
        return p;
}
```

1.1.3 Обратная польская запись

a) Вычисление
b) Получение обратной польской записи из исходного выражения (алгоритм Дейкстры).

1. Операнд – в выходную строку
2. Отрывающая скобка – а стек
3. Знак – в стек с выталкиванием в выходную строку знаков с большим приоритетом до отрывающей скобки
4. Закрывающая скобка – выталкиваем из стека в выходную строку все знаки до открывающей скобки. Обе скобки выбрасываем
5. Конец входной стоки – стек в выходную строку

Просматриваемый символ	1	2	3	4	5	6	7	8	9	10	11	12	13	
Входная строка	(	A	+	B	)	*	(	C	+	D	)	-	E	
Состояние стека	((	((	+ ((	+ ((		*	(* *	(* *	+ (* *	+ (* *	*	-	-	
Выходная строка		A		B	+			C		D	+	*	E	-

1.1.4 Сортировка вставками

1.1.5 Самоорганизующийся список (программа с фиктивным элементом)

```
struct List{
	int key;
	List * next;
};

List* CreateList(void)
{
	int i;
	List *p=NULL, *q;

	for (i=10; i>0; i--)
	{
		q=(List*)malloc(sizeof(List));
		q->key=i;
		q->next=p;
		p=q;
	}
	q=(List*)malloc(sizeof(List));
	q->next=p;
	return q;
}

void PrintList(List *s)
{
	s=s->next;
	while (s!=NULL)
	{
		printf("%d ", s->key);
		s=s->next;
	}
   printf( "\n" );
}

List* Search(List *s, int key)
{
	List *p1, *p2;

	p2=s;
	p1=s->next;
	while (p1!=NULL)
	{
		if (p1->key==key)
			break;
		p2=p1;
		p1=p1->next;
	}
	if (p1!=NULL)
	{
```

```
		p2->next=p1->next;
		p1->next=s->next;
		s->next=p1;
	}
	return p1;
}

int main(void)
{
	List *s=CreateList(), *q;
	PrintList(s);
	q=Search(s,5);
	if (q==NULL)
		printf("Element is absent\n");
	else
		printf("%d\n", q->key);
	PrintList(s);
	getch();
	return 0;
}
```

1.1.6 Подсчет числа вхождений слов в файле (программа с барьером)

```
struct List{
	char *word;
	int count;
	List *next;
};

static List* Last;

List* InitList(void)
{
	List *s;

	Last=s=(List*)malloc(sizeof(List));
	s->next=NULL;
	return s;
}

List* Add2List(List* s, char* word)
{
	List *q=s;

	Last->word=word;
	while (strcmp(q->word,word))
		q=q->next;
	if (q!=Last)
		q->count++;
	else
	{
		q=(List*)malloc(sizeof(List));
```

```
                q->word=word;
                q->count=1;
                q->next=s;
                s=q;
        }
        return s;
}

void PrintList(List *s)
{
        while (s!=Last)
        {
                printf("%s %d\n", s->word, s->count);
                s=s->next;
        }
}

int main(void)
{
        List *s=NULL;
        FILE *f;
        char buf[80];
        char *word;

        if( (f = fopen( "wordlist.txt", "r" )) == NULL )
                printf( "The file 'WordList' was not opened\n" );
        s=InitList();
        fscanf(f, "%s", buf);
        while (!feof(f))
        {
                word = (char *)malloc(strlen(buf)+1);
                strcpy(word, buf);
                s=Add2List(s, word);
           fscanf(f, "%s", buf);
        }

        PrintList(s);
        getch();
        return 0;
}
```

1.1.7 Сложение двух полиномов

```
struct List{
      int coef;
      int pow;
      List *next;
};

struct Queue{
      List *first, *last;
};

void InitQueue(Queue *q)
```

```
{
	q->first=q->last=(List*)malloc(sizeof(List));
}

void Add2Queue(Queue *q, int coef, int pow)
{
	List *p;

	p=(List*)malloc(sizeof(List));
	p->coef=coef;
	p->pow=pow;
	p->next=NULL;
	q->last->next=p;
	q->last=p;
}

void ReadPol(FILE *f, Queue *q)
{
	char buf[6];
	int coef, pow;

	do {
	    fscanf(f,"%s", buf);
	      coef=atoi(buf);
	    fscanf(f,"%s", buf);
	      pow=atoi(buf);
	      Add2Queue(q, coef, pow);
	} while(!feof(f));
}
void PrintPol(Queue *q)
{
	List *p=q->first;
	while (p!=q->last)
	{
		p=p->next;
		printf("%d %d ", p->coef, p->pow);
	}
	printf("\n");
}

void SupPol(Queue *q1, Queue *q2, Queue *q3)
{
	List *p1=q1->first->next, *p2=q2->first->next;
	int s;

	while (p1!=NULL && p2!=NULL)
	{
		if (p1->pow > p2->pow)
		{
			Add2Queue(q3, p1->coef, p1->pow);
			p1=p1->next;
		}
		else
		{
			if (p1->pow < p2->pow)
			{
				Add2Queue(q3, p2->coef, p2->pow);
				p2=p2->next;
			}
			else
			{
				s=p1->coef+p2->coef;
				if (s!=0)
				{
```

```
                                Add2Queue(q3, s, p1->pow);
                                p1=p1->next;
                                p2=p2->next;
                        }
                }
            }
        }
        while (p1!=NULL)
        {
            Add2Queue(q3, p1->coef, p1->pow);
            p1=p1->next;
        }

        while (p2!=NULL)
        {
            Add2Queue(q3, p2->coef, p2->pow);
            p2=p2->next;
        }
}

int main(void)
{
        FILE *f;
        Queue q1, q2, q3;

        if( (f = fopen( "pol1.txt", "r" )) == NULL )
            printf( "The file 'pol1' was not opened\n" );
        InitQueue(&q1);
        ReadPol(f,&q1);
        fclose(f);
        PrintPol(&q1);

        if( (f = fopen( "pol2.txt", "r" )) == NULL )
            printf( "The file 'pol2' was not opened\n" );
        InitQueue(&q2);
        ReadPol(f,&q2);
        fclose(f);
        PrintPol(&q2);

        InitQueue(&q3);
        SupPol(&q1, &q2, &q3);
        PrintPol(&q3);

        getch();
        return 0;
}
```

1.2 Управление динамической памятью

Списки свободного пространства

1.2.1 Блоки одинакового размера

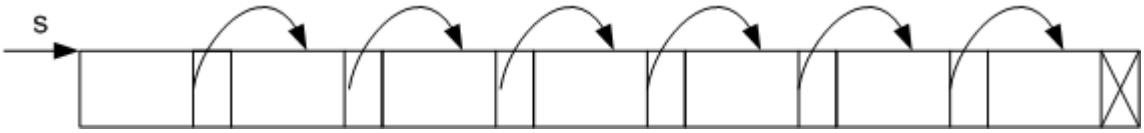


Рис. 1.1

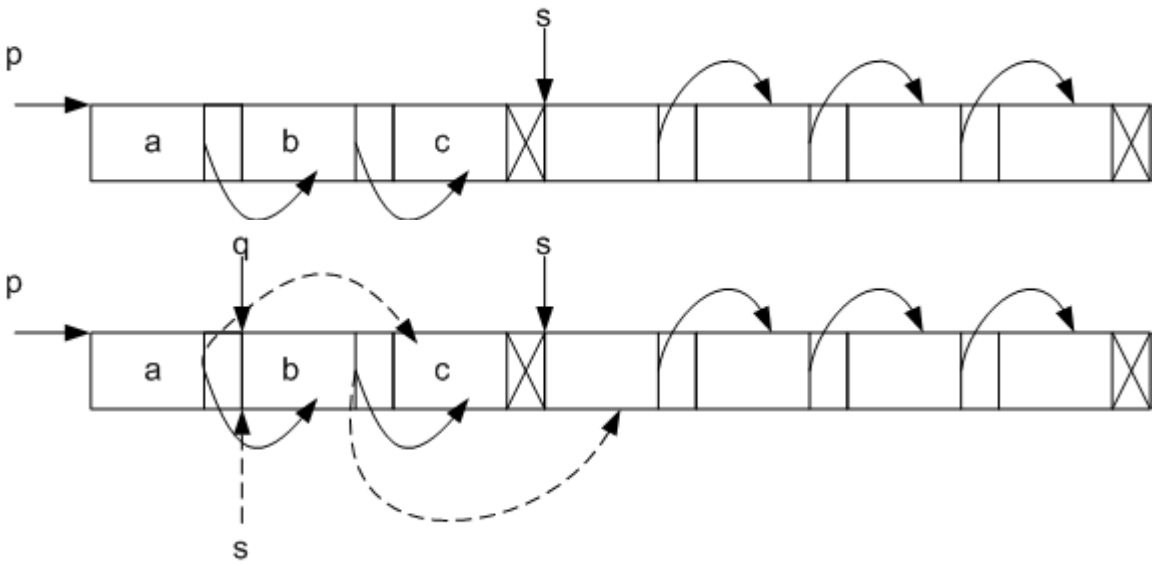


Рис. 1.2

1.2.2 Блоки произвольного размера

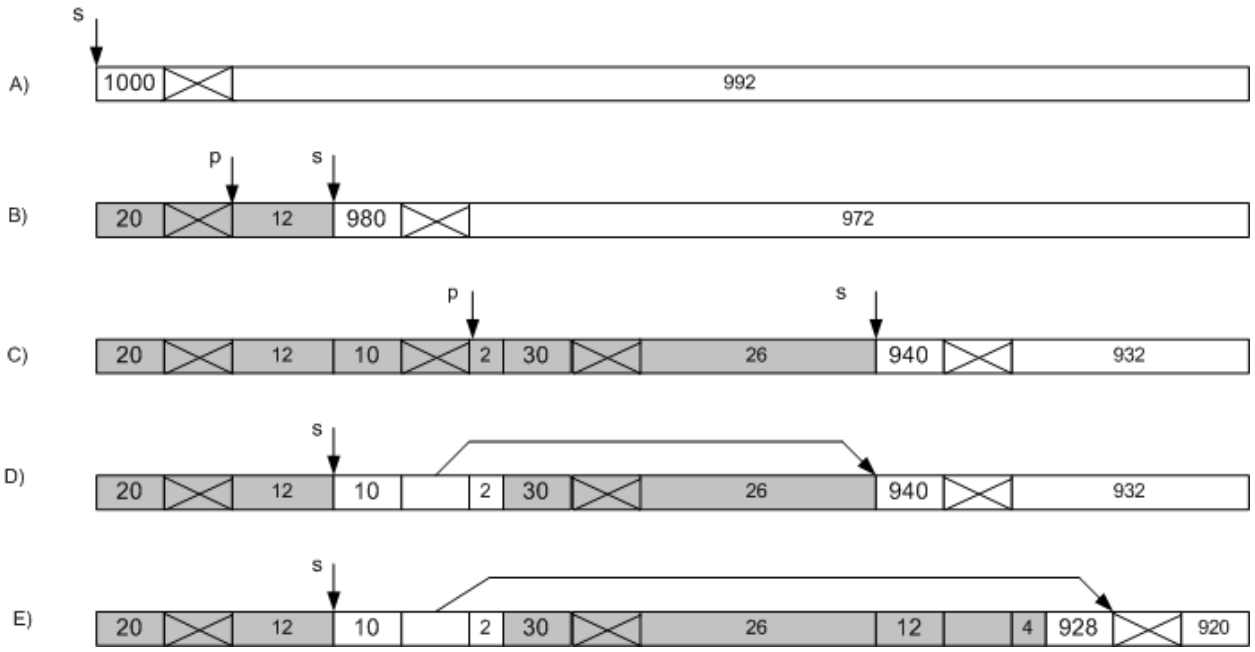


Рис. 1.3

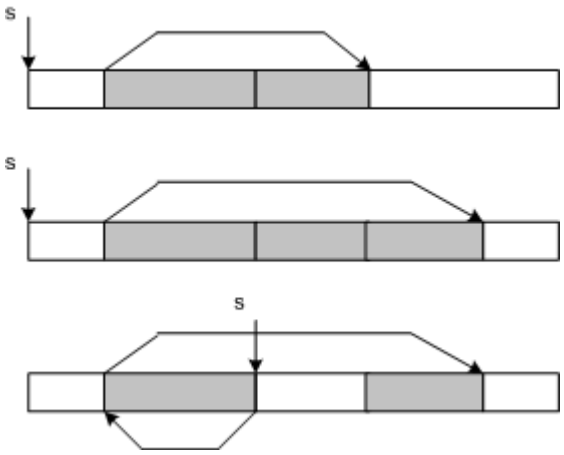


Рис. 1.4 Фрагментация

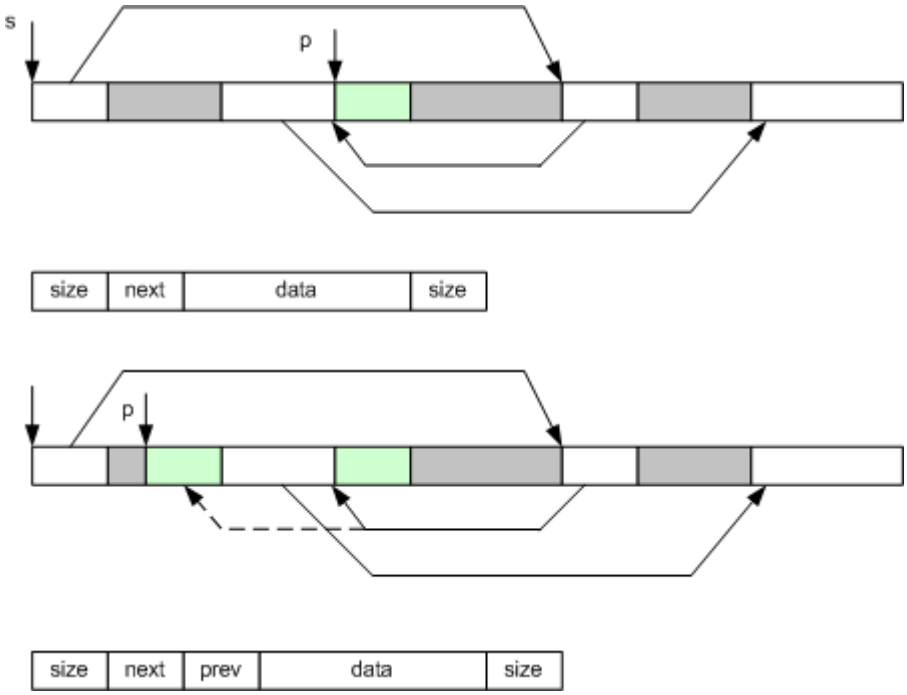


Рис. 1.5 Объединение

Поиск подходящего блока:

- Первый подходящий (first fit)
- Ближайший по размеру (best fit)

Кольцевой список для исключения скопления мелких блоков в начале списка
Проверка целостности списка.
Контроль повторного освобождения блока.

1.3 Структуры со списками

1.3.1 Записная книжка

1.3.2 Представление матриц

1.3.3 Растеризация невыпуклого многоугольника

1.3.4 Топологическая сортировка

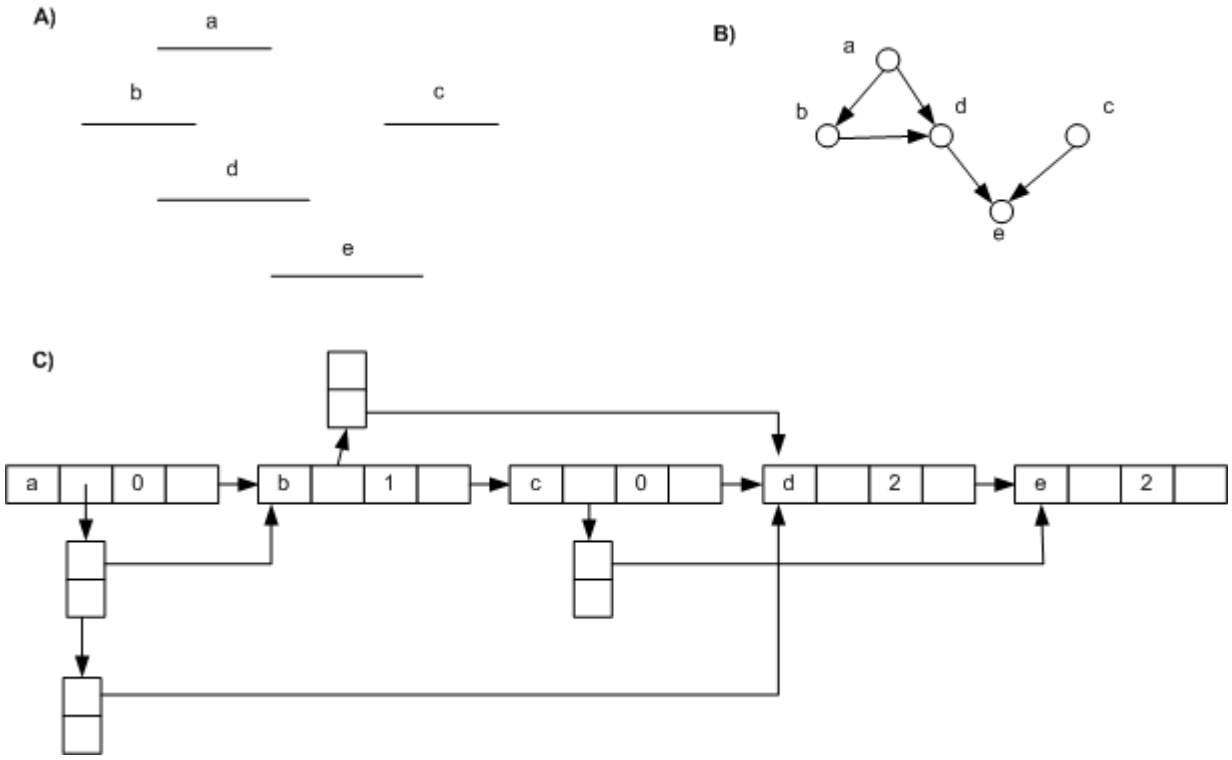


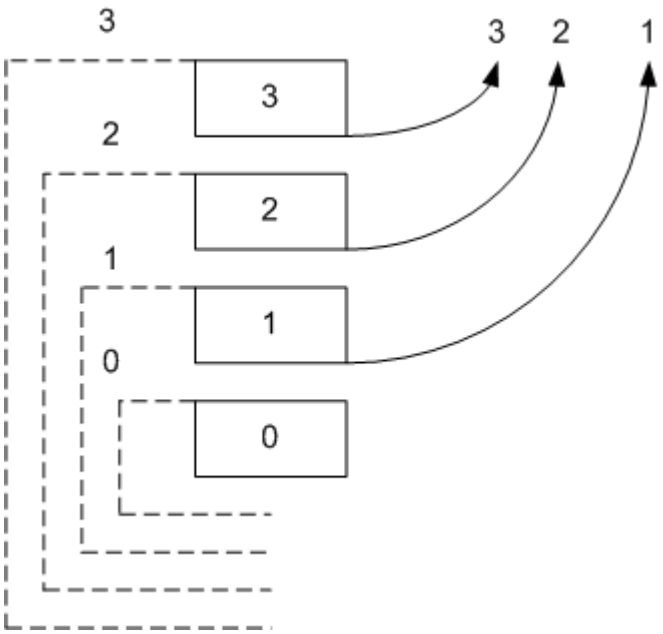
Рис. 1.6

2 Рекурсия

2.1 Процедуры с одним рекурсивным вызовом

2.1.1 Порядок обработки данных

```
void Print1(int n)
{
        if (n>0)
        {
                printf("%d",n);
                Print1(n-1);
        }
}
Print1(3)
```




```
void Print2(int n)
{
        if (n>0)
        {
                Print2(n-1);
                printf("%d",n);
        }
}
```

Print2(3)

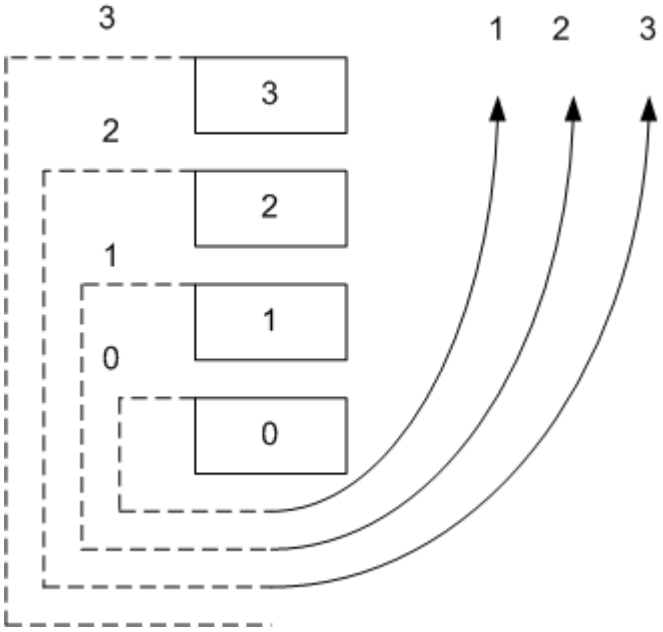


Рис. 2.1

2.1.2 Неоправданное использование рекурсии

Факториал

```
int f(int n)
{
	if (n==0)
		return 1;
	else
		return n*f(n-1);
}
```

2.1.3 Оправданное использование рекурсии

Обращение входного потока

```
void Reverse(void)
{
	char *ch=getch();

	if (ch!='.')
	{
		Reverse();
		putch(ch);
	}
}
```

2.2 Процедуры с двумя рекурсивными вызовами

2.2.1 Порядок обработки данных

```
void Print1(int n)
{
	if (n>0)
	{
		printf("%d",n);
		Print1(n-1);
		Print1(n-1);
	}
}
Print1(3)
```

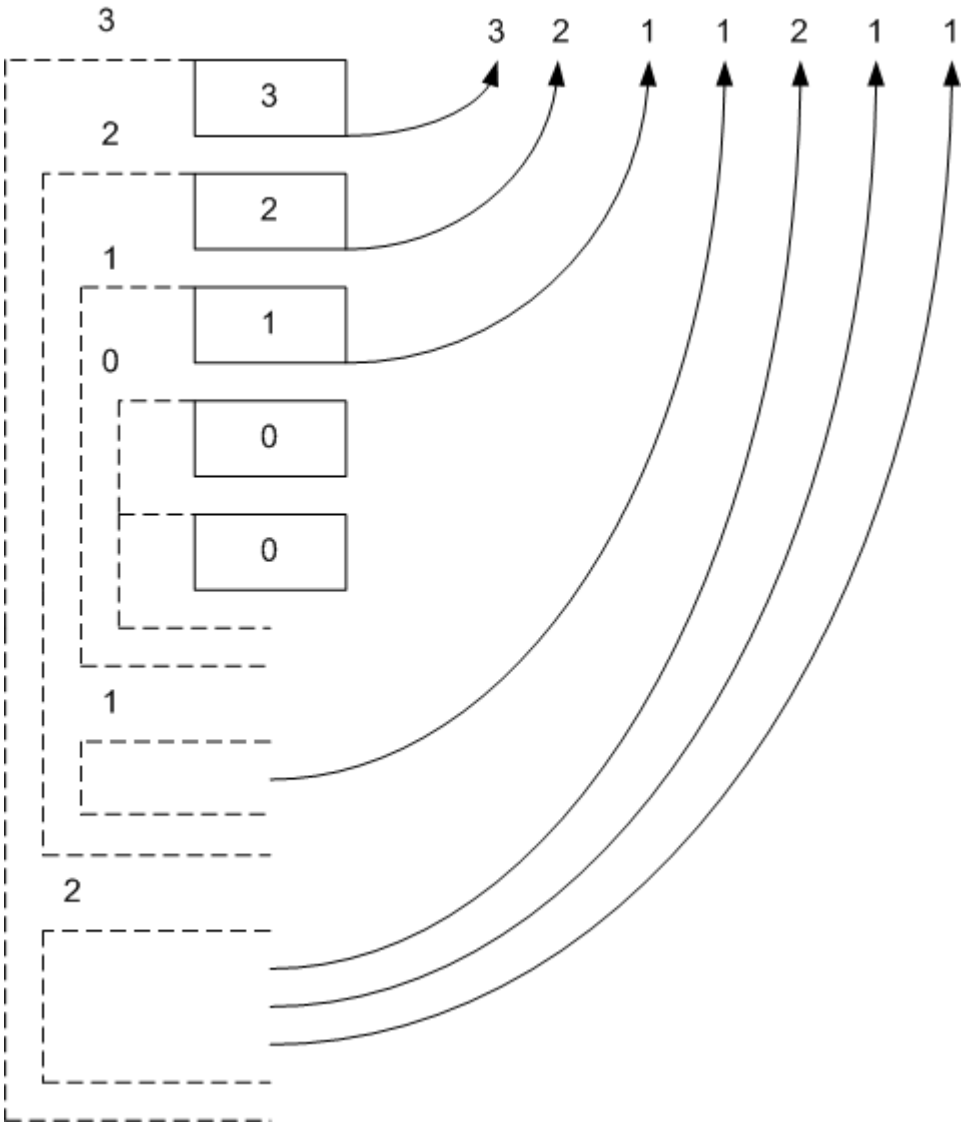


Рис. 2.2

```
Print1(0)=;
Print1(1)=1;
Print1(2)=211;
Print1(3)=3211211;

void Print2(int n)
{
        if (n>0)
        {
                Print2(n-1);
                printf("%d",n);
                Print2(n-1);
        }
}
void Print3(int n)
{
        if (n>0)
        {
                Print3(n-1);
```

```
                Print3(n-1);
                printf("%d",n);
        }
}
```

2.2.2 Задача о “ханойской башне”

```
void Hanoy(int n, char x, char y, char z)
{
        if (n>0)
        {
                Hanoy(n-1,x,z,y);
                printf("%c to %c ",x,y);
                Hanoy(n-1,z,y,x);
        }
}
void main(void)
{
   Hanoy(3,'x','y','z');
   getch();
}
```

Сложность:

$$T(n)=\begin{bmatrix}1, n=1\\ 2T(n-1)+1, n>1\end{bmatrix}$$

$T(n)=2(2T(n-2)+1)+1=2^2T(n-2)+2+1=\ldots 2^nT(n-n)+2^{n-1}+\ldots+2+1$

$T(n)=2^n+2^{n-1}+\ldots+2+1=2^{n+1}+1=O(2^n)$

2.2.3 Синтаксический анализ

Expression::=letter | (Expression Sign Expression)

```
char ch;
void Expression(FILE *f)
{
  if (ch>='a' &&  ch<='z')
    ch=fgetc(f);
  else
  {
    if (ch=='(')
    {
      ch=fgetc(f);
      Expression(f);
      if (ch == '+' || ch == '-' || ch == '*' || ch == '/')
      {
          ch=fgetc(f);
          Expression(f);
          if (ch==')')
            ch=fgetc(f);
          else
```

```
            exit(1);
        }
        else
            exit(1);
    }
    else
        exit(1);
  }
}
int main()
{
  FILE *f;

  if ((f = fopen("Expres.txt", "rt")) == NULL)
  {
     printf("Cannot open Expres.txt\n");
     return 1;
  }
  ch=fgetc(f);
  Expression(f);
  printf("OK\n");
  fclose(f);
  return 0;
}
```

2.2.4 Сортировка слияниями

```
void sort(int i, int j)
{
        int k;
        if (i<j)
        {
                k=(i+j)>>1;
                sort(i,k);
                sort(k+1,j);
                merging(i,k,j);
        }
}
```

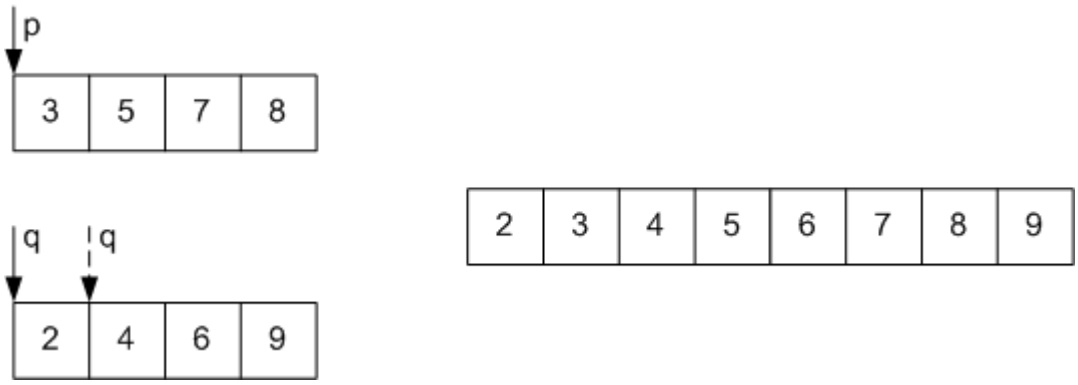


Рис. 2.3

```
        merging(n)=O(n)
```

$$T(n) = \begin{bmatrix} 1, n = 1 \\ 2T(n/2) + n, n > 1 \end{bmatrix}$$

$T(n)=2(2T(n/4)+n/2)+n=2^2T(n/2^2)+2n=\ldots 2^kT(n/2^k)+kn$
$n=2^k$
$T(n) =n+kn=n+\log(n)n =O(n\log(n))$

2.2.5 Числа Фибоначи

```
int fib(int n)
{
        if (n==0)
                return 0;
        else
        {
                if (n==1)
                        return 1;
                else
                        return fib(n-1)+fib(n-2);
        }
}
int fib(int n)
{
        if (F[n]>=0)
                return F[n];
        else
                return F[n]=fib(n-1)+fib(n-2);
}

x=0;
y=1;
for (int i=0; i<n; i++)
{
        z=y+x;
        x=y;
}
```

2.3 Косвенная рекурсия

Кривые Гильберта

A: D<-A | A->B
B: C | B->B | A
C: B->C | C<-D
D: A | D<-D | C

```
void A(int n)
{
        if (n>0)
        {
```

```
                D(n-1); line(...);
                A(n-1); line(...);
                A(n-1); line(...);
                B(n-1);
        }
}
void B(int n)
{
        if (n>0)
        {
                C(n-1); line(...);
                B(n-1); line(...);
                B(n-1); line(...);
                A(n-1);
        }
}
void C(int n)
{
        if (n>0)
        {
                B(n-1); line(...);
                C(n-1); line(...);
                C(n-1); line(...);
                D(n-1);
        }
}
void D(int n)
{
        if (n>0)
        {
                A(n-1); line(...);
                D(n-1); line(...);
                Dn-1); line(...);
                C(n-1);
        }
}
```

2.4 Рекурсия в цикле

2.4.1 Перебор

```
char a[n];
for (int i0=0; i0<n; i0++)
{
        a[0]=буква(i0);
        for (int i1=0; i1<n; i1++)
        {
                a[1]=буква(i1);
                for (int i2=0; i2<n; i2++)
```

```
                    a[2]=буква(i2);
            ......
                    for (int i_{n-1}=0; i_{n-1}<n; i_{n-1}++)
                    {
                            a[i_{n-1}]=буква(i_{n-1});
                            Print (a);
                    }
            ......
        }
}
void per(int k)
{
        a[k]='a';
        while (a[k]<'c')
        {
                if (k==n-1)   printA(a);
                else    per(k+1);
                a[k]++;
        }
}
void main()
{
  per(0);
}
```

2.4.2 Перестановки

1. <1, 2, 3, 4>	7. <2, 1, 3, 4>	13. <3, 1, 2, 4>	19. <4, 1, 2, 3>
2. <1, 2, 4, 3>	8. <2, 1, 4, 3>	14. <3, 1, 4, 2>	20. <4, 1, 3, 2>
3. <1, 3, 2, 4>	9. <2, 3, 1, 4>	15. <3, 2, 1, 4>	21. <4, 2, 1, 3>
4. <1, 3, 4, 2>	10. <2, 3, 4, 1>	16. <3, 2, 4, 1>	22. <4, 2, 3, 1>
5. <1, 4, 2, 3>	11. <2, 4, 1, 3>	17. <3, 4, 1, 2>	23. <4, 3, 1, 2>
6. <1, 4, 3, 2>	12. <2, 4, 3, 1>	18. <3, 4, 2, 1>	24. <4, 3, 2, 1>

<15, 2, 4, 3, 1, 13, 7, 10, 14, 12, 11, 9, 8, 6, 5>,
<15, 2, 4, 3, 1, 13, 7, 11, 5, 6, 8, 9, 10, 12, 14>.

```
void Lec(int k)
{
  int i;
  if (k==n-1)
      print p;
  else
      for (i=n-1; i>=k; i--)
        {
              Lec(k+1);
              if (i>k)
              {
                    p[i]<->p[k];
                    Invert(k+1);
              }
      }
}
void main()
{
    for (int i=0; i<n; i++)p[i]=i;
      Lec(0);
}
```

2.4.3 Синтаксический анализ

```
Expression::= Exp{знак Exp }
Exp::=x|(Expression)
x::=целое
знак::=+ | - | * | /

FILE *f;
char ch;
void Exp(void);
void Expression(void)
{
  Exp();
  while (ch == '+' || ch == '-' || ch == '*' || ch == '/')
  {
    ch=fgetc(f);
    Exp();
  }
}
void Exp(void)
{
  if (ch>='a' &&  ch<='z')
    ch=fgetc(f);
  else
  {
    if (ch=='(')
    {
      ch=fgetc(f);
      Expression();
      if (ch==')')
           ch=fgetc(f);
      else
           exit(1);
    }
    else
      exit(1);
  }
}

int main()
{
……................
  ch=fgetc(f);
  Expression();
……................
}

выражение::= слагаемое{знак1 слагаемое}
слагаемое::=множетель{знак2 множетель}
множетель::=x|( выражение)
x::=целое
```

знак1::=+ | -
знак2::=* | /

2.5 Сокращение перебора

2.5.1 N ферзей.

```
procedure полный_перебор(m : integer);
 var i : integer;
 begin
     if m > n  then
        <проверка построенной позиции>
     else
        for i := 1  to n  do begin
          ферзь[m] := i;
          полный_перебор(m+1);
        end;
 end;
```

Алгоритм с возвратом

```
 procedure перебор_с_возвратом(m : integer);
 var i : integer;
 begin
     if m > n  then
       <найдено решение>
     else
        for i := 1  to n  do begin
           ферзь[m] := i;
           if <ферзь[m] не бьёт предыдущих>  then
               перебор_с_возвратом(m+1);
        end;
 end;
```

2.5.2 Покрыть шахматное поле ходами коня.

а) Одно решение

```
void TryNextMove()
{
        while (нет решения && есть ходы)
        {
                выбрать ход;
                if (ход возможен)
                {
                        запомнить ход;
                        if (доска заполнена)
                                есть решение;
                        else
                        {
                                TryNextMove();
                                if (нет решения)  забыть ход;
                        }
```

```
                }
        }
}
int q=0;
void TryNextMove(int n, int x, int y)
{
        int u, v, m=0;

        while (q==0 && m<8)
        {
                u=x+a[m];
                v=y+b[m];
                m++;
                if ((u>-1)&&(u<8)&&(v>-1)&&(v<8)&&(h[u][v]==0))
                {
                        h[u][v]=n;
                        if (n==64)
                                q=1;
                        else
                        {
                                TryNextMove(n+1,u,v);
                                if (q==0)
                                        h[u][v]=0;
                        }
                }
        }
}
```

b) Все решения

```
void TryNextMove()
{
        while (есть ходы)
        {
                выбрать ход;
                if (ход возможен)
                {
                        запомнить ход;
                        if (доска заполнена)
                                выдать решение;
                        else
                        {
                                TryNextMove();
                                 забыть ход;
                        }
                }
        }
}
```

2.5.3 Задача о коммивояжере

```
void коммивояжер(int k, вершина v)
{
      for (w соседих с v)
```

```
        {
                if {L+C[v,w]<Lmin}
                {
                        if (w==начальный узел и k==n)
                        {
                                Lmin=L+C[v,w];
                                Smin <- S;
                        }
                        else
                        {
                                if (T[w]==0)
                                {
                                        S[k]=w;
                                        T[w]=1;
                                        L=L+C[v,w];
                                        коммивояжер(k+1, w);
                                        T[w]=0;
                                        L=L-C[v,w];
                                }
                        }
                }
        }
}
```

2.5.4 Использование симметрии

Раскраска карты

```
procedure перебор_с_возвратом(m : integer);
var i : integer;
begin
  if m > n  then
     <найдено решение>
  else
     for i := 1  to k  do begin
        цвет[m] := i;
        if <цвета стран 1,...,m-1 соседних с m-ой не i>  then
         перебор_с_возвратом(m+1);
        end;
end;


procedure перебор_с_возвратом(m : integer);
var i : integer;
begin
   if m > n  then
      <найдено решение>
   else begin
       for i := 1  to кол_красок  do begin
          цвет[m] := i;
          if <цвета стран 1,...,m-1 соседних с m-ой не i>  then
              перебор_с_возвратом(m+1);
       end;
       if кол_красок < k  then begin
          кол_красок := кол_красок+1;
          цвет[m] := кол_красок;
          перебор_с_возвратом(m+1);
          кол_красок := кол_красок-1;
       end;
   end;
end;
```

2.5.5 Распространение ограничений

					3	2	1
				3	2	1	
			3	2	1		
		3	2	1	3	3	3
		2	1				
	2	1	2	2	2	2	2
	1	2			3		
1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8

Рис. 2.4

```
procedure распространение_ограничений(m : integer);
var i : integer;
   if m > n  then
       <найдено решение>
   else
       for i := 1  to n  do
            if пространство[m,i] = 0  then begin
                ферзь[m] := i;
                сократить_пространство_перебора(m,i);
                распространение_ограничений(m+1);
                восстановить_пространство_перебора(m,i);
            end;
end;
```

2.5.6 Изменение порядка перебора

					3	2	1
				3	2	1	
			3	2	1		4
		3	2	1	3	3	3
		2	1	4	**4**	4	4
	2	1	2	2	2	2	2
	1	2	4		3		4
1	1	1	1	1	1	1	1
1	2	3	4	5	6	7	8

Рис. 2.5

2.5.7 Метод ветвей и границ

Укладка рюкзака. Из заданных *n* предметов выбрать такие, чтобы их суммарная стоимость была не менее чем *S*, а суммарный объём не превосходил *V*.

Неизвастна длина результата.

```
procedure перебор_с_возвратом(m : integer);
var i : integer;
begin
   if m > n  then
       <найдено решение>
   else begin
      x[m] := 0;
      потерянная_стоимость := потерянная_стоимость+стоимость[m];
      if полная_стоимость - потерянная_стоимость => S  then
          перебор_с_возвратом(m+1);
      потерянная_стоимость := потерянная_стоимость-стоимость[m];
      x[m] := 1;
      сумм_объём := сумм_объём+объём[m];
      if сумм_объём <= V  then
          перебор_с_возвратом(m+1);
      сумм_объём := сумм_объём-объём[m];
   end;
end;
```

3 Деревья

Определения

3.1 Бинарные деревья

3.1.1 Представление деревьев в памяти

a. Хранение двух указателей

```
struct tree{
    char ch;
    tree * left;
    tree * right;
};
```

b. Паралельные массивы
c. Хранение в одном массиве

3.1.2 Обходы

a. Обходы в прямом, внутреннем и обратном порядке

```
void PrintTree(tree *t)
{
    if (t!=NULL)
    {
            printf("%c",t->ch);
```

```
            PrintTree(t->left);
            PrintTree(t->right);
    }
}
void PrintTree(tree *t)
{
    if (t!=NULL)
    {
            PrintTree(t->left);
            printf("%c",t->ch);
            PrintTree(t->right);
    }
}
void PrintTree(tree *t)
{
    if (t!=NULL)
    {
            PrintTree(t->left);
            PrintTree(t->right);
            printf("%c",t->ch);
    }
}
```

b. Раскрытие рекурсии

```
tree *st[10];
int s=0;
void PrintTree(tree *t)
{
    s++; st[s]=t;
    while (s)
    {
            t=st[s]; s--;
            while (t!=NULL)
            {
                    printf("%c",t->ch);
                    s++; st[s]=t->right;
                    t=t->left;
            }
    }
}

struct stackT{
    int type;
    tree *t;
};
stackT stk[20];
int s=0;

void PrintTree(tree *t)
{
    int type;

    s++; stk[s].type=1; stk[s].t=t;
```

```
    while (s)
    {
        t=stk[s].t; type=stk[s].type; s--;
        if (t!=NULL)
        {
            if (type==0)
                printf("%c",t->ch);
            else
            {
                s++; stk[s].type=1; stk[s].t=t->right;
                s++; stk[s].type=0; stk[s].t=t;
                s++; stk[s].type=1; stk[s].t=t->left;
            }
        }
    }
}
```

c. Обход в ширину

```
void PrintTree(tree *t)
{
    S<-t;
    while (S не пусто)
    {
        t<-S;
        if (t!=NULL)
        {
            printf("%c",t->ch);
            S<- t->right;
            S<- t->left;
        }
    }
}

void PrintTree(tree *t)
{
    Q<-t;
    while (Q не пусто)
    {
        t<-Q;
        if (t!=NULL)
        {
            printf("%c",t->ch);
            Q<- t->left;
            Q<- t->right;
        }
    }
}
```

3.1.3 Чтение, сохранение и печать деревьев

a. Сохранение в файле

```
void SaveTree(FILE *f, tree *t)
```

```
{
    if (t!=NULL)
    {
            fputc(t->ch,f);
            if (t->left)
                    fputc('+',f);
            else
                    fputc('-',f);

            if (t->right)
                    fputc('+',f);
            else
                    fputc('-',f);

            SaveTree(f,t->left);
            SaveTree(f,t->right);
    }
}
```

b. Чтение из файла

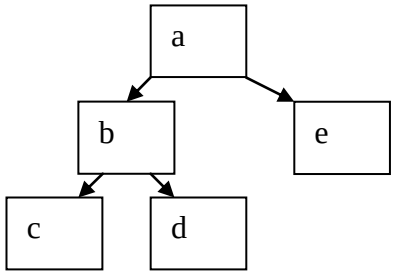
```
tree *ReadTree(FILE *f)
{
    char ch, ch_l, ch_r;
    tree *t;

    ch=fgetc(f);
    ch_l=fgetc(f);
    ch_r=fgetc(f);
    if (!feof(f))
    {
            t = (tree *)malloc(sizeof(tree));
            t->ch=ch;
            t->left=NULL;
            t->right=NULL;
            if (ch_l != '-')
                    t->left=ReadTree(f);
            if (ch_r != '-')
                    t->right=ReadTree(f);
    }
    return t;
}
```

c. Печать

```
void PrintTree(tree *t, int n)
{
    int i;
    if (t!=NULL)
    {
            PrintTree(t->right, n+1);
            for (i=0; i<n; i++) putchar(' ');
            printf("%c\n",t->ch);
            PrintTree(t->left, n+1);
    }
}
```

```
x=1
void PrintTree(tree *t, int y)
{
    if (t!=NULL)
    {
            PrintTree(t->left, y+1);
            gotoxy(x,y);  // A[x][y]=t;
            printf("%c",t->ch);
            x++;
            PrintTree(t->right, y+1);
    }
}
```

d. Быстрое чтение



a	1	4	b	2	3	c	0	0	d	0	0	e	0	0

0 1 2 3 4

3.1.4 Самоорганизующаяся экспертная система

3.1.5 Счетчики

a) Число узлов

```
int n=0;
void Count(tree *t)
{
    if (t!=NULL)
    {
            n++;
            Count(t->left);
            Count(t->right);
    }
}
int Count(tree *t)
{
    int r1, r2;
    if (t==NULL) return 0;
    else
    {
            r1=Count(t->left); // t->nl = r1
            r2=Count(t->right); // t->nr = r1
```

```
            return (r1+r2+1);
    }
}
```

b) Высота

```
int h=0;
int CountH(tree *t, int n)
{
    if (t!=NULL)
    {
            if (t->left==NULL && t->right==NULL)
            {
                    if (n>h) h=n;
            }
            CountH(t->left, n+1);
            CountH(t->right, n+1);
    }
}
CountH(t, 0);

int CountH(tree *t)
{
    int r1, r2, r;
    if (t==NULL) return 0;
    else
    {
            if (t->left==NULL && t->right==NULL) return 0;
            else
            {
                    r1=CountH(t->left); // t->nl = r1
                    r2=CountH(t->right); // t->nr = r2
                    if (r1>r2) r=r1+1;
                    else r=r2+1;
                    return r;
            }
    }
}
```

3.2 Сильноветвящиеся деревья и графы

Представление в памяти

```
Tree:
    Int key;
    Tree* brother;
    Tree* child;
```

Обходы

a) Дерево

Q – stack или очередь

```
Q<-t;
```

```
While (Q!=пусто)
{
    t<-Q; обработать(t);
    t=t->child;
    while (t!=NULL)
    {
            Q<-t; t=t->brother;
    }
}
```

b) Граф

```
Q – stack или очередь
for( всех u)
    u->flag=0;
u->flag=1;
Q<-u;
While (Q!=пусто)
{
    u<-Q;
    обработать(u);
    for (v связанных с u)
    {
            if (v->flag==0)
            {
                    Q<-v;
                    v->flag=1;
            }
    }
}
```

3.3 Деревья двоичного поиска

```
struct tree{
        int key;
        tree * left;
        tree * right;
};

tree* Add(tree *t, int x)
{
        if (t==NULL)
        {
                t=(tree *)malloc(sizeof(tree));
                t->key=x;
                t->left=t->right=NULL;
        }
        else
        {
                if (x<t->key)
```

```
                        t->left=Add(t->left,x);
                else
                {
                        if (x>t->key)
                                t->right=Add(t->right,x);
                }
        }
        return t;
}


tree* del1(tree* &q, tree *p)
{
        tree *r;

        if (q->right!=NULL)
          r=del1(q->right,p);
        else
        {
                p->key=q->key;
                r=q;
                q=q->left;
        }
        return r;
}

tree* del(tree *t, int x)
{
  tree *p;

  if (t!=NULL)
  {
        if (x<t->key)
          t->left=del(t->left,x);
        else
        {
          if (x>t->key)
                t->right=del(t->right,x);
          else
          {
                p=t;
                if (p->right==NULL)
                  t=p->left;
                else
                {
                  if (p->left==NULL)
                        t=p->right;
                  else
                        p=del1(p->left,p);
                }

                free(p);
```

```
            }
        }
    }
    return t;
}
```

3.4 Деревья с дополнительной информацией

3.4.1 Добавление с поддержкой числа узлов

```
struct tree{
        int key;
        int number; //число узлов в поддереве
        tree* left;
        tree* right;
};

int add(tree* &t, int k)
{
        int flag;

        if(t == NULL)
        {
                t=(tree1*)malloc(sizeof(tree1));
                t->key = k;
                t->left = t->right = NULL;
                t-> number = 1;
                flag=1;
        }
        else
        {
                if(k < t->key)
                {
                        flag = add(t->left, k);
                        if (flag)
                                t->number += 1;
                }
                else
                {
                        if(k > t->key)
                        {
                                flag = add(t->right, k);
                                if (flag)
                                        t->number += 1;
                        }
                        else
                                flag=0;
                }
        }
        return flag;
}
```

3.4.2 К-й наименьший

```
tree* k_min (tree* t, int k)
{
	int  r;

	if (t->left)
	{
		r=t->left->number + 1;
	}
	else
		r=1;
	if (r!=k)
	{
		if (k<r)
			t=k_min(t->left,k);
		else
			t=k_min(t->right,k-r);
	}
	return t;
}
```

3.4.3 Порядковый номер

```
struct tree{
	int key;
	int number; //число узлов в поддереве
	tree* left;
	tree* right;
	tree* father;
};
int Number(tree *t)
{
	tree *f;
	int r = Size(t); //t->left->number + 1;

	while (t!=root)
	{
		f=t->father;
		if (t==f->right)
			r+=Size(f->left)+1;
		t=t->father;
	}
	return r;
}
```

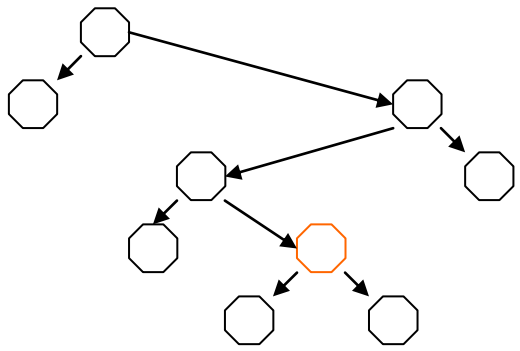


Рис. 3.1

3.4.4 Пересекающиеся отрезки

```
struct tree{
	int l;
	int r;
	int r_max;
	tree* left;
	tree* right;
};
tree* Intersect(tree *t, int l, int r)
{
	tree *p;
	while (t!=NULL && (t->l > r || t->r < l))
	{
		p=t->left;
		if (p!=NULL && (p->r_max > l))
			t=t->left;
		else
			t=t->right;
	}
	return t;
}
```

3.4.5 BSP-tree

3.4.6 Oct-tree

4 Динамическое программирование

4.1 Конвейер

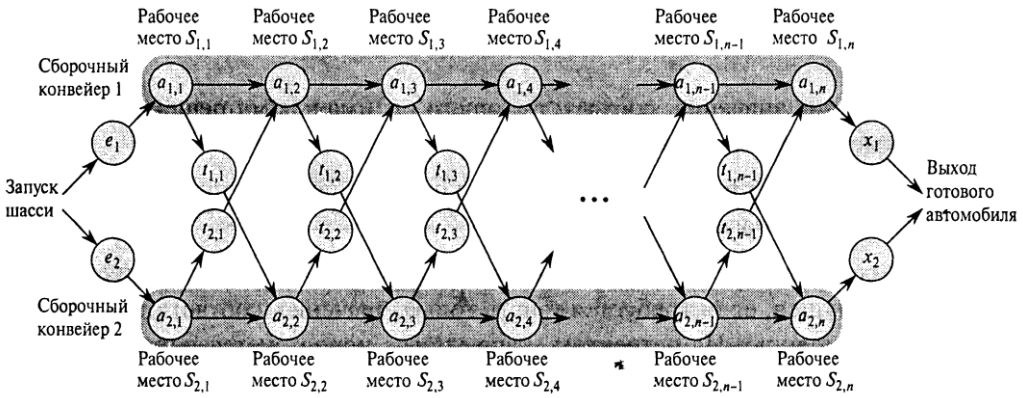


Рис. 4.1

S_{1j-1} , S_{2j-1} => S_{ij}

Опимальное решение задачи строится из отимальных решений подзадач – оптимальная подструктура.

Разделяй и властвуй – независимые подзадачи. Оптимальная подструктура - зависимые

f_i [j] минимальное время до S_{ij}.

$$f^* = \min(f_1[n] + x_1, f_2[n] + x_2).$$

$$f_1[j] = \begin{cases} e_1 + a_{1,1} & \text{при } j = 1, \\ \min(f_1[j-1] + a_{1,j}, f_2[j-1] + t_{2,j-1} + a_{1,j}) & \text{при } j \geqslant 2. \end{cases}$$

$$f_2[j] = \begin{cases} e_2 + a_{2,1} & \text{при } j = 1, \\ \min(f_2[j-1] + a_{2,j}, f_1[j-1] + t_{1,j-1} + a_{2,j}) & \text{при } j \geqslant 2. \end{cases}$$

$T_1[n] = T_2[n] = T_1[n-1] + T_2[n-1] + 1=$
$= (T_1[n-2] + T_2[n-2] + 1)+ (T_1[n-2] + T_2[n-2] + 1) + 1= 2T_1[n-2] + 2T_2[n-2]+2+1=$
$= 2^kT_1[n-k] + 2^kT_2[n-k]+2^{k-1}+\ldots+1=O(2^n)$

Вычисление минимального времени

```
f1[1] ← e1 + a1,1
f2[1] ← e2 + a2,1
for j ← 2 to n
    do if f1[j − 1] + a1,j ≤ f2[j − 1] + t2,j−1 + a1,j
          then f1[j] ← f1[j − 1] + a1,j
               l1[j] ← 1
          else f1[j] ← f2[j − 1] + t2,j−1 + a1,j
               l1[j] ← 2
       if f2[j − 1] + a2,j ≤ f1[j − 1] + t1,j−1 + a2,j
          then f2[j] ← f2[j − 1] + a2,j
               l2[j] ← 2
          else f2[j] ← f1[j − 1] + t1,j−1 + a2,j
               l2[j] ← 1
if f1[n] + x1 ≤ f2[n] + x2
    then f* = f1[n] + x1
         l* = 1
    else f* = f2[n] + x2
         l* = 2
```

Вычисление минимального пути

```
i ← l*
print "Конвейер " i ", рабочее место " n
for j ← n downto 2
    do i ← l_i[j]
        print "Конвейер " i ", рабочее место " j − 1
```

4.2 Наименьшая сумма чисел на пути

7
3 8
8 1 0
2 7 4 4
4 5 2 6 5

Каждый шаг на пути - вниз и влево или вниз и вправо.

$a_{i,j}$ - j-ое число в i-ой строке, $s_{i,j}$ – наименьшая сумма чисел от вершины до числа $a_{i,j}$.

$$s_{1,1} = a_{1,1},\ s_{i,1} = a_{i,1} + s_{i-1,1},\ s_{i,i} = a_{i,i} + s_{i-1,i-1},$$
$$s_{i,j} = a_{i,j} + min\ \{s_{i-1,j}, s_{i-1,j-1}\},\ 1 < j < i \le n$$

```
s[1,1]=a[1,1];
for (i=2; i<n; i++)
{
      s[i,1]=a[i,1]+s[i-1,1]);
      s[i,i]=a[i,i]+s[i-1,i-1]);
      for (j=2; j<i; j++)
            s[i,j]=a[i,j]+min(s[i-1,j],s[i-1,j-1]);
}
```

MinS = $\min_j$(s[n,j])

7.3 Умножение матриц

A*B p*q q*r => p*q*r
$A_1A_2A_3$ 10*100, 100*5, 5*50
$(A_1A_2)A_3$ - 7500
$A_1(A_2A_3)$ - 75000

$A_iA_{i+1}\ldots A_j$
$A_i \ldots A_k\ A_{k+1}\ldots A_j$
A_i - $p_{i-1}p_i$
m[i,j] = m[i,k]+ m[k+1,j]+$p_{i-1}p_kp_j$

m[i,j] = $\min_k$(m[i,k]+ m[k+1,j]+$p_{i-1}p_kp_j$) i<j
(m[i,i]=0)
$O(2^n)$, $O(n^3)$

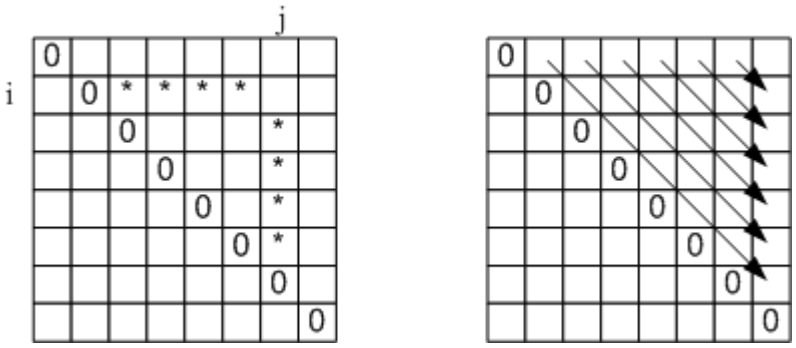


Рис. 4.2

$A_{1.} \ldots A_k$ и $A_{k+1} \ldots A_j$ не подходит

```
LOOKUP_CHAIN(p, i, j)
1  if m[i, j] < ∞
2      then return m[i, j]
3  if i = j
4      then m[i, j] ← 0
5      else  for k ← i to j − 1
6                do q ← LOOKUP_CHAIN(p, i, k)
                        + LOOKUP_CHAIN(p, k + 1, j) + p_{i−1} p_k p_j
7                   if q < m[i, j]
8                      then m[i, j] ← q
9  return m[i, j]
```

В строке 8 добавить s[i,j]=k
Тогда
$A_{1.} \ldots A_{k1}$ и $A_{k1+1} \ldots A_n$ где k1=s[1,n]
$A_{1.} \ldots A_{k2}$ и $A_{k2+1} \ldots A_{k1}$ где k2=s[1,k1]

```
Скобки(i,j)
{
        if (i==j ) print "Aj";
        else
        {
                print "(" ;
                Скобки(i,s[i,j]);
                Скобки(s[i,j]+1,j);
                print ")" ;
        }
}
```

4.3 Оптимальная триангуляция многоугольника

$v_0, v_1, \ldots, v_{n-1}$

$$w(\Delta v_i v_k v_j) = |v_i v_k| + |v_k v_j| + |v_i v_j|$$

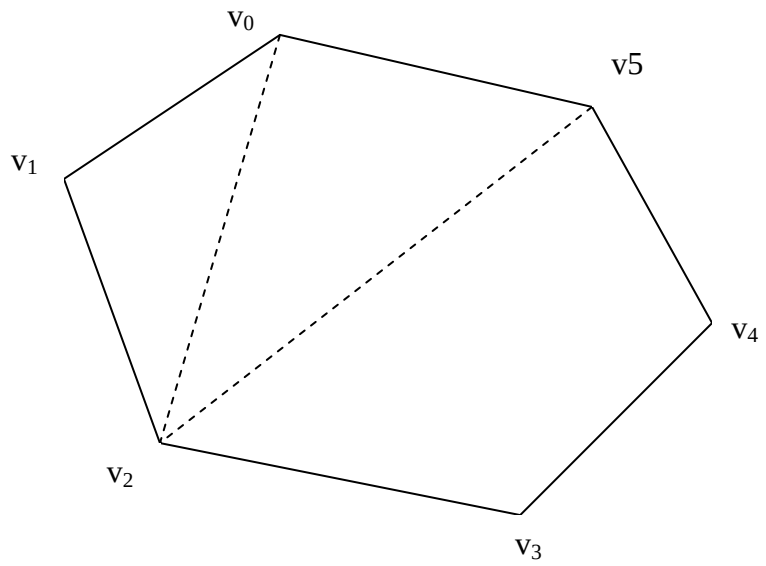


Рис. 4.3

m[i,j] = 0 i=j
m[i,j] = min $_{i<=k<j}$ {m[i,k] + m[k+1,j] + $w(\Delta v_{i-1} v_k v_j)$ } i<j,

4.4 Самая длинная общая подпоследовательность

X = (x_1, x_2, • • •, x_m) и Y = (y_1, y_2, • • •, y_n)
Z = (z_1, z_2, • • •, z_k)
X = (A, B, C, B, D, A, B) и Y = (B, D, C, A, B, A)
Z = (B, C, B, A)

1. if (x_m = y_n)
   z_k = x_m = y_n => Z_{k-1} max из X_{m-1} и Y_{n-1}
   else
   {
   if (z_k <> x_m) => Z_k max из X_{m-1} и Y_n
   else => Z_k max из X_m и Y_{n-1}
   }

$$c[i,j] = \begin{cases} 0 & \text{при } i = 0 \text{ или } j = 0, \\ c[i-1, j-1] + 1 & \text{при } i, j > 0 \text{ и } x_i = y_j, \\ \max(c[i, j-1], c[i-1, j]) & \text{при } i, j > 0 \text{ и } x_i \neq y_j. \end{cases}$$

```
LCS_LENGTH(X, Y)
m ← length[X]
n ← length[Y]
for i ← 1 to m
      do c[i, 0] ← 0
for j ← 0 to n
      do c[0, j] ← 0
for i ← 1 to m
      do for j ← 1 to n
              do if x_i = y_j
                    then c[i, j] ← c[i − 1, j − 1] + 1
                         b[i, j] ← "↖"
                    else if c[i − 1, j] ≥ c[i, j − 1]
                            then c[i, j] ← c[i − 1, j]
                                 b[i, j] ← "↑"
                            else c[i, j] ← c[i, j − 1]
                                 b[i, j] ← "←"
return c и b
```

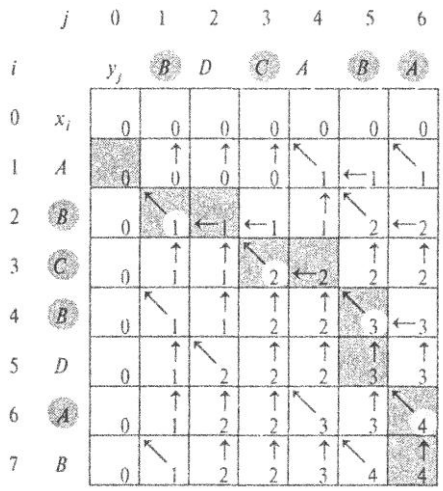


Рис. 4.4

5 Сбалансированные деревья

5.1 Идеально сбалансированные деревья

Построение идеально сбалансированного дерева

tree* Ideal(int n)

```
{
        tree* t;
        int k;

                t=(tree*)malloc(sizeof(tree));
                t->key=a[i++];
                t->left=t->right=NULL;

                k=n/2;
                if (k>0)
                        t->left=Ideal(k);
                if (n-k-1>0)
                        t->right=Ideal(n-k-1);

        return t;
}
```

5.2 АВЛ дерево

Балансировка

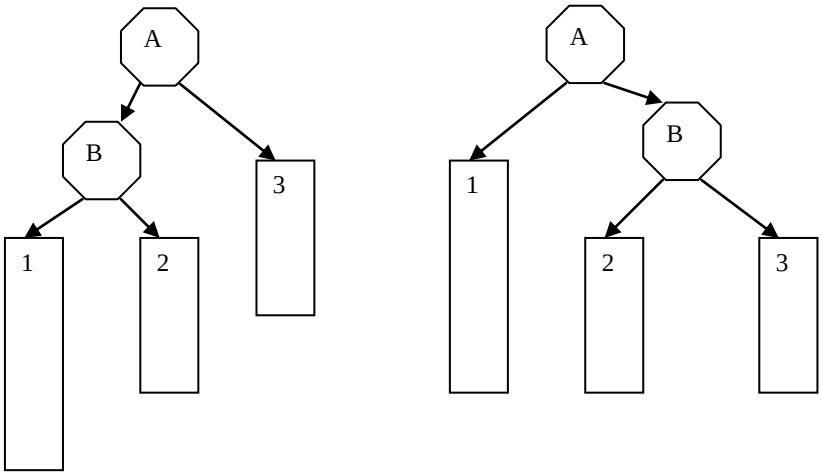


Рис. 5.1

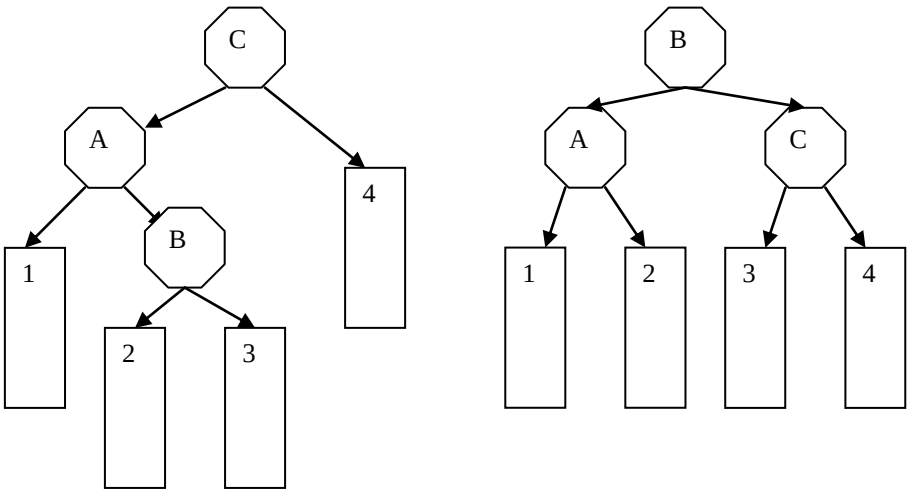


Рис. 5.2

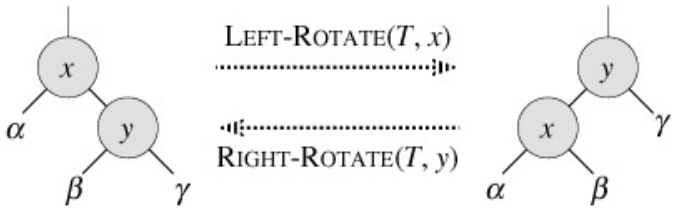


Рис. 5.3

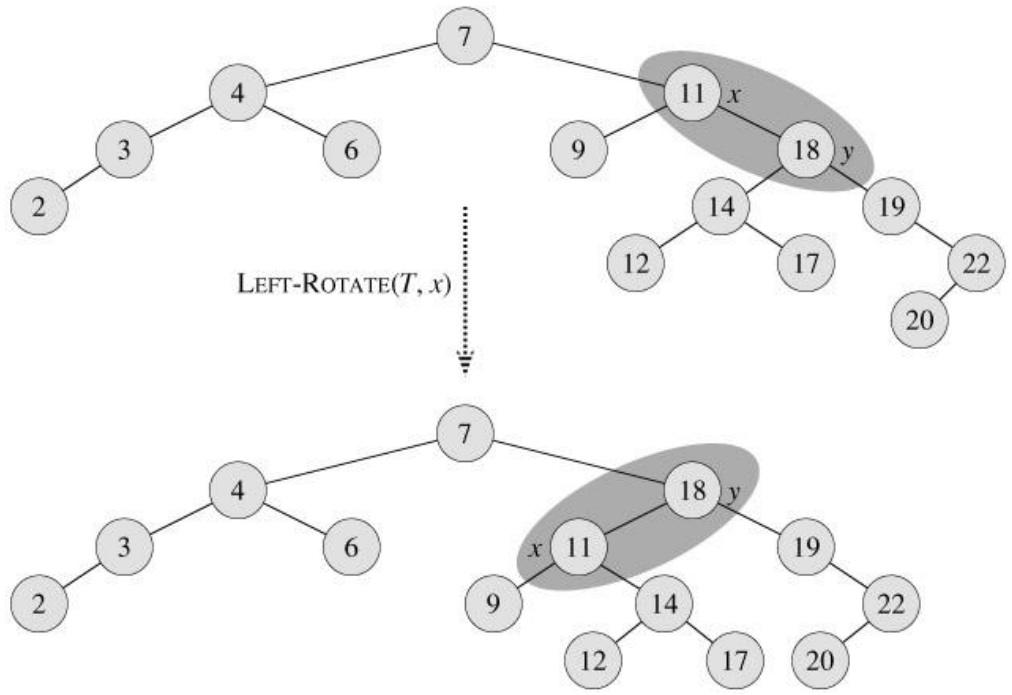


Рис. 5.4

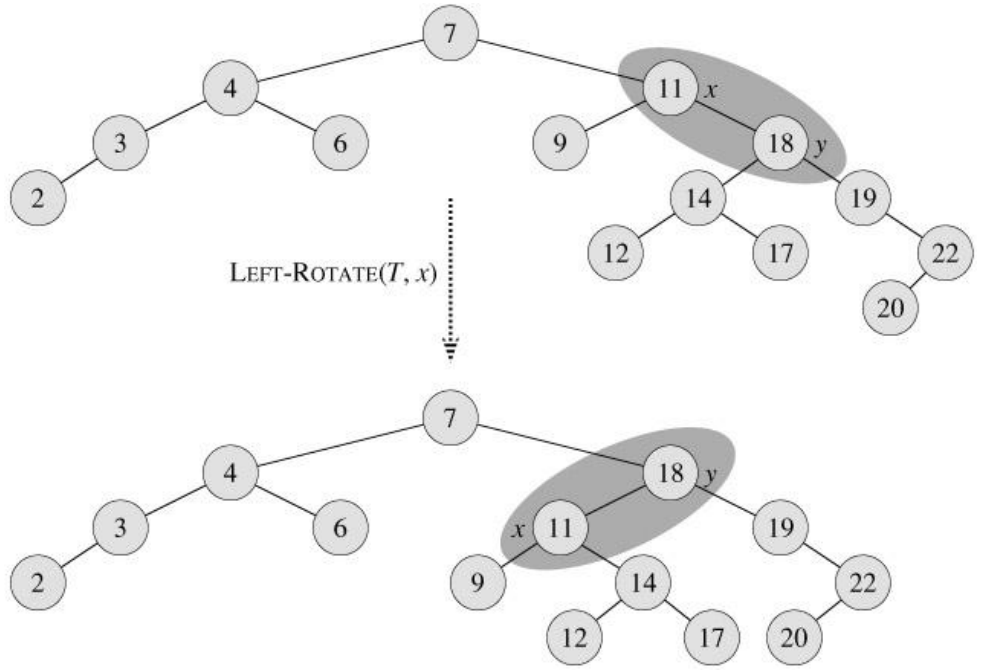


Рис. 5.5

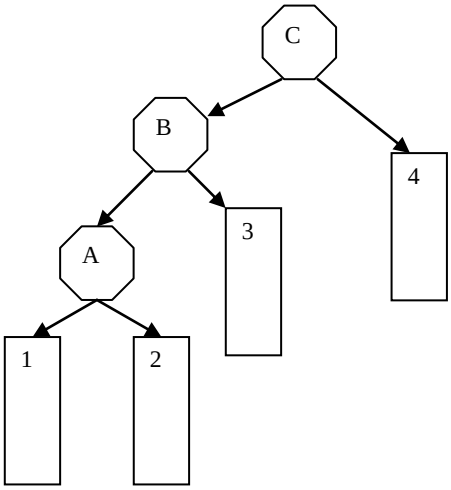


Рис. 5.6

Добавление

```
struct_tree{
        int key;
        int bal;
        tree * left;
        tree * right;
};

void Add(tree* &t, int x, int flag)
{
        tree * t1;

        if (t==NULL)
        {
                t=malloc(...);
                ...
                t->bal=0;
                flag=1;
        }
        else
        {
                if (x<t->key)
                {
                        Add(t->left,x,flag);
                        if (flag)
                        {
                                switch (t->bal)
                                case  1: t->bal=0; flag=0; break;
                                case  0: t->bal=-1; break;
                                case  -1:  t1=t->left;
                                        if (t1->bal=-1)
```

```
                    {
                        t->left=t1->right;
                        t1->right=t;
                        t->bal=0;
                        t=t1;
                    }
                    else  два поворота;
            }
```

Удаление

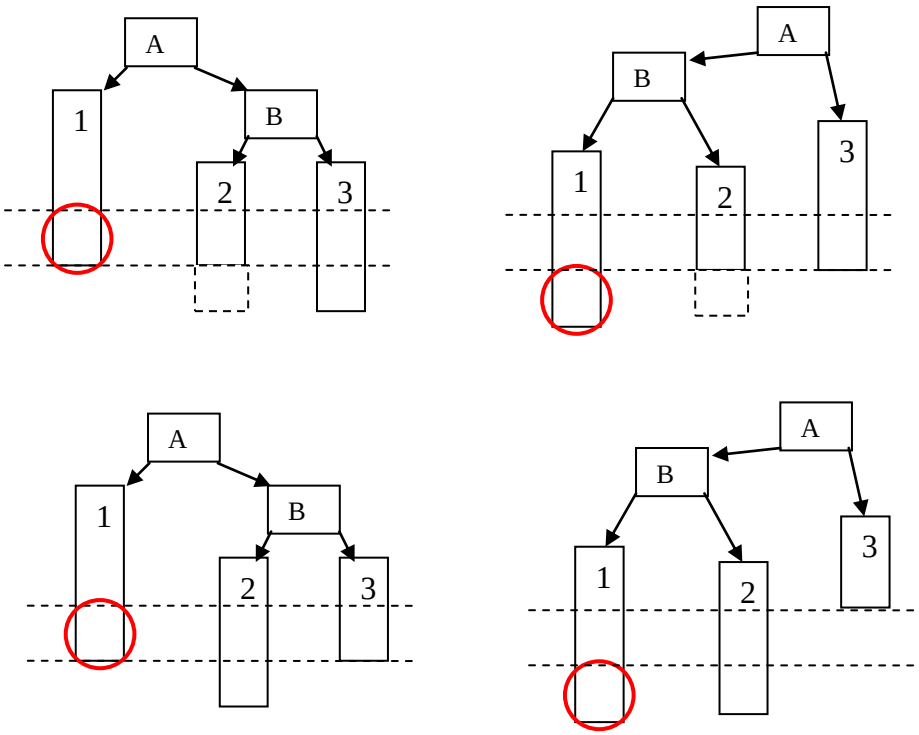


Рис. 5.7

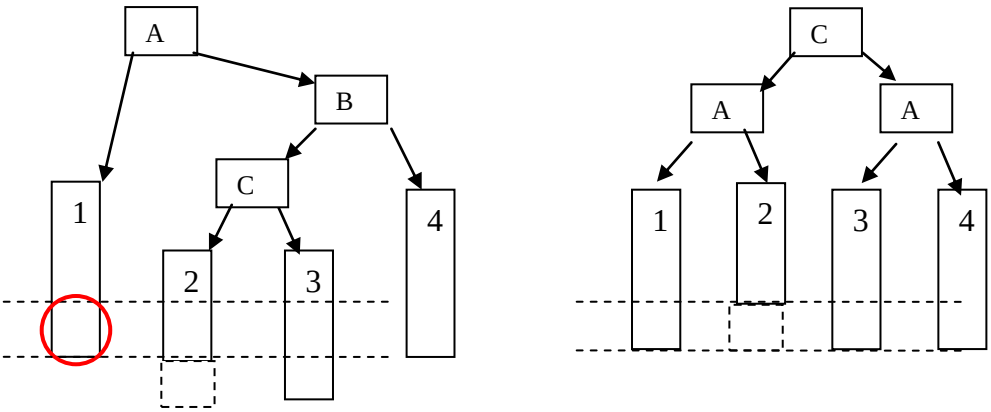


Рис. 5.8

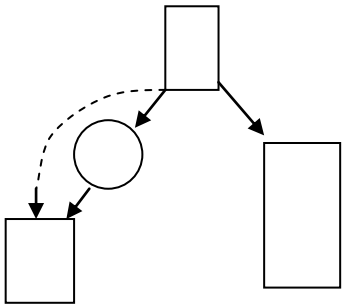


Рис. 5.9

5.3 Красно-черное дерево

Определение

1. Каждый узел является красным или черным.
2. Корень дерева является черным.
3. Каждый лист дерева (nil) является черным.
4. Если узел — красный, то оба его дочерних узла — черные.
5. Для каждого узла все пути от него до листьев, являющихся потомками данного узла, содержат одно и то же количество черных узлов.

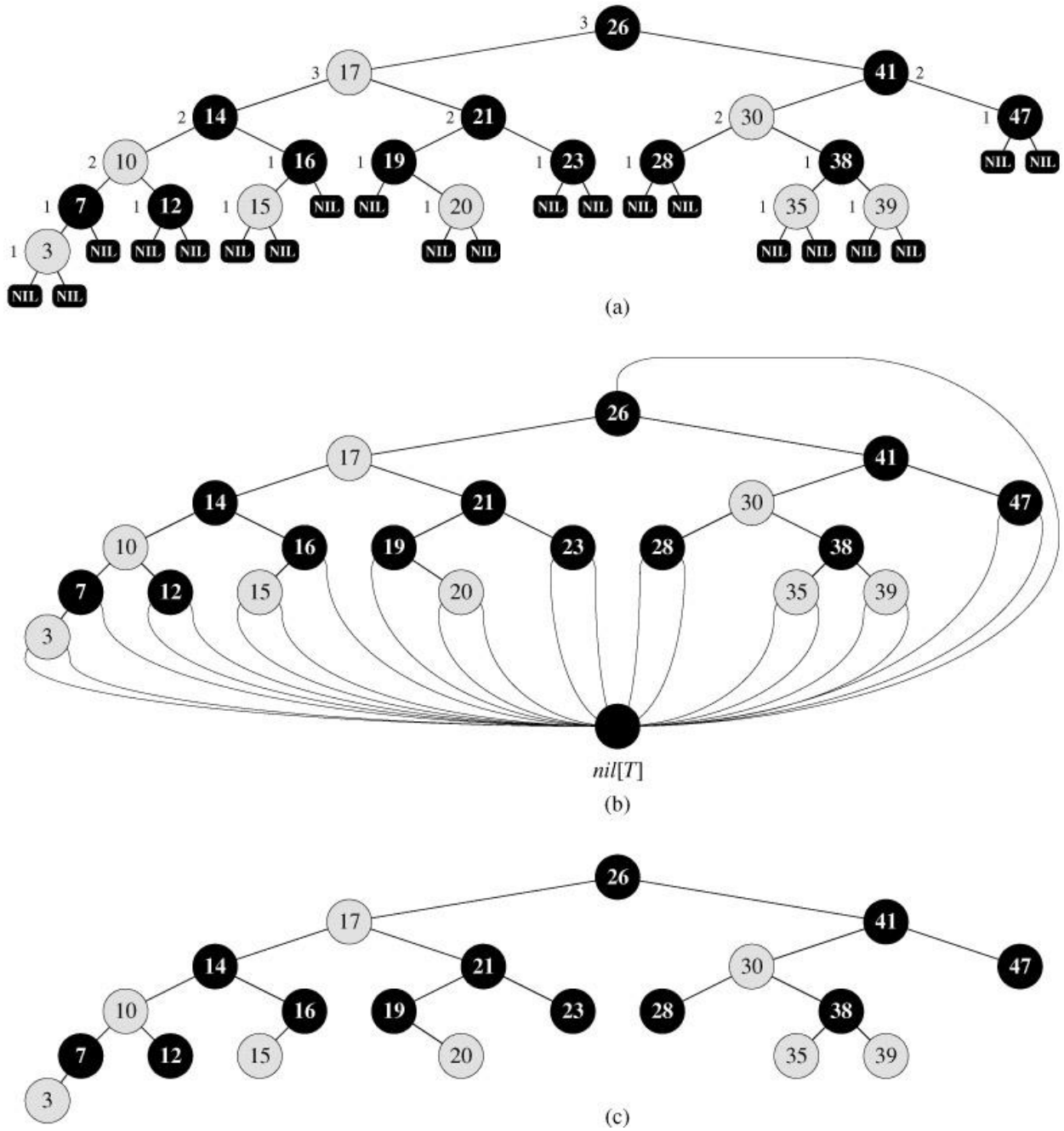


Рис. 5.10

Балансировка красно-черного дерева после добавления

```
RB-INSERT-FIXUP(T, z)
 1 while color[p[z]] = RED
 2     do if p[z] = left[p[p[z]]]
 3           then y ← right[p[p[z]]]
 4                if color[y] = RED
 5                   then color[p[z]] ← BLACK                    ▷ Case 1
 6                        color[y] ← BLACK                       ▷ Case 1
 7                        color[p[p[z]]] ← RED                   ▷ Case 1
 8                        z ← p[p[z]]                            ▷ Case 1
 9                   else if z = right[p[z]]
10                           then z ← p[z]                      ▷ Case 2
11                                LEFT-ROTATE(T, z)             ▷ Case 2
```

```
                        color[p[z]] ← BLACK                    ▷ Case 3
                        color[p[p[z]]] ← RED                   ▷ Case 3
                        RIGHT-ROTATE(T, p[p[z]])               ▷ Case 3
          else (same as then clause
                    with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

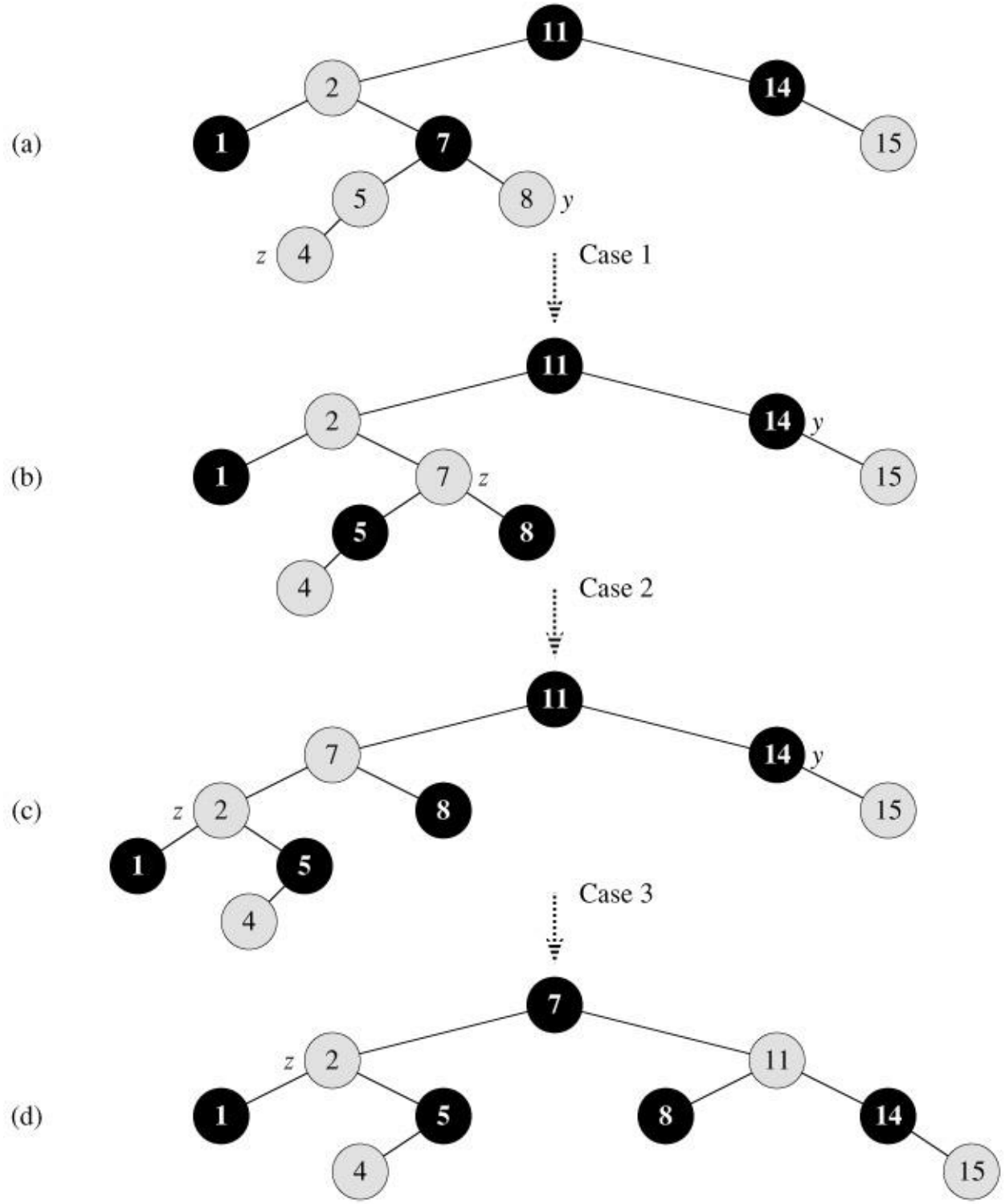


Рис. 5.11

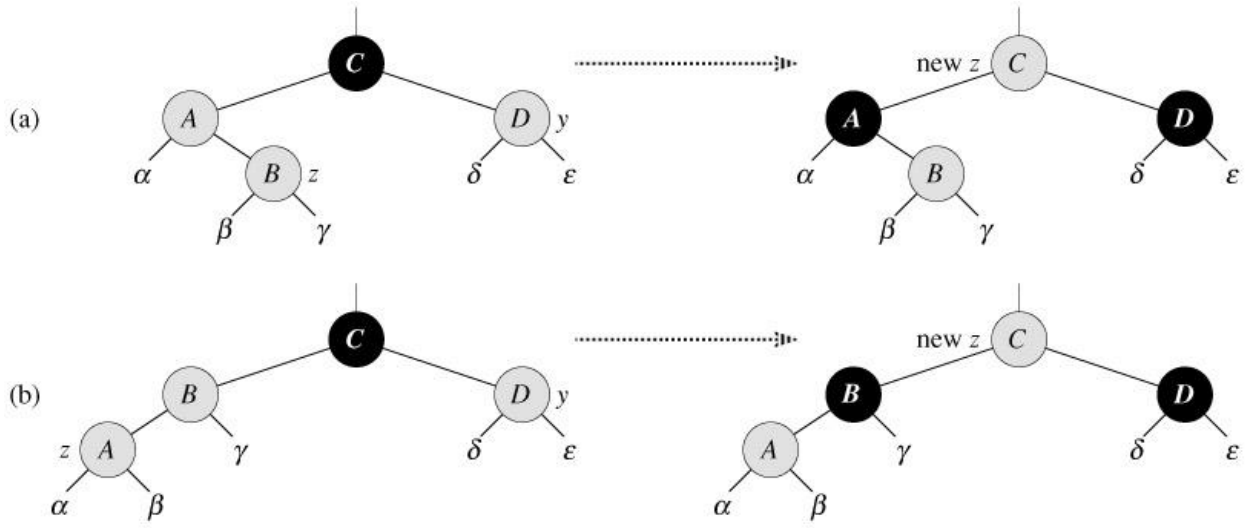


Рис. 5.12

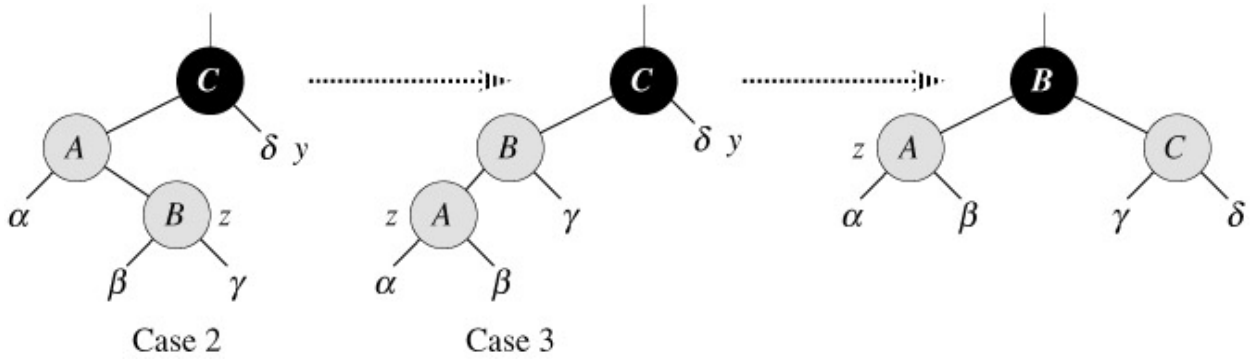


Рис. 5.13

Удаление

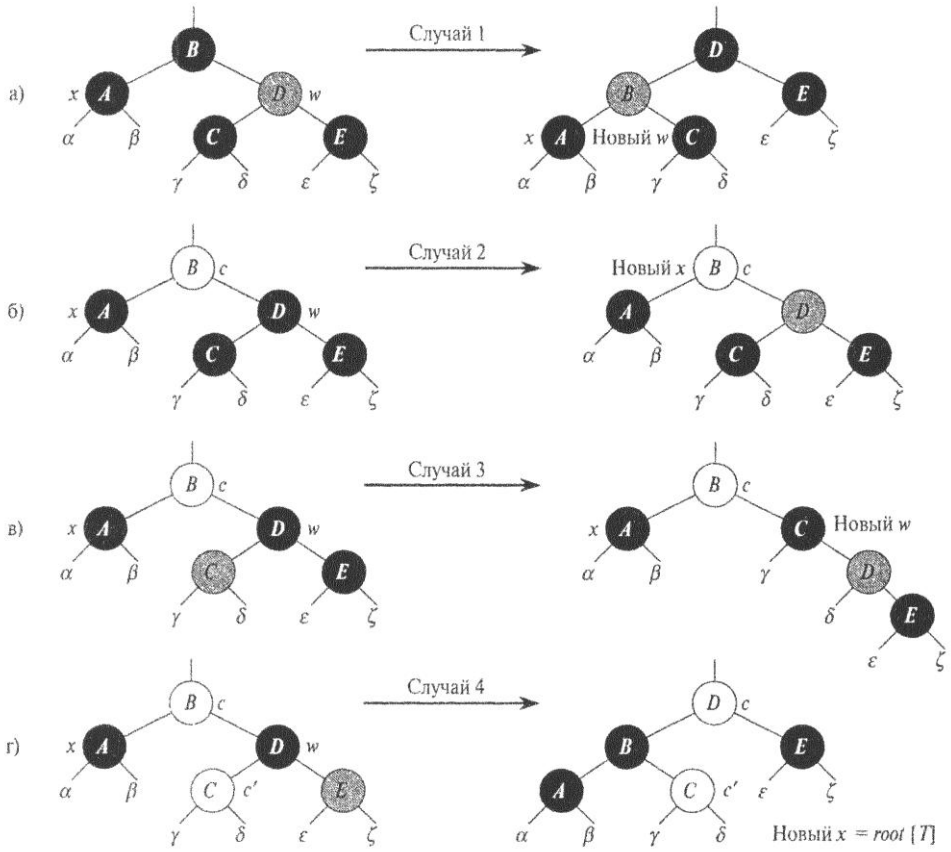


Рис. 5.14

5.4 B-tree

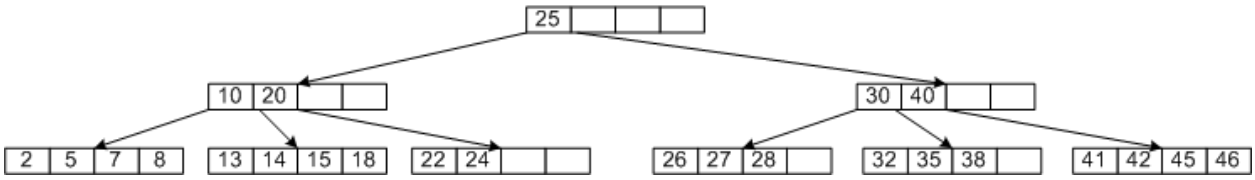


Рис. 5.15

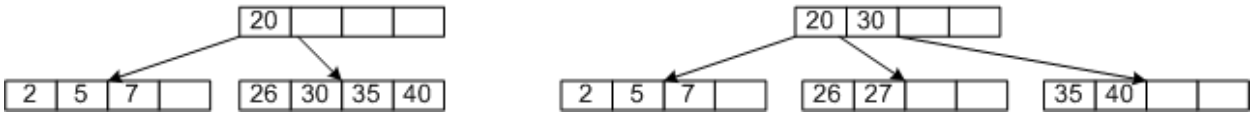
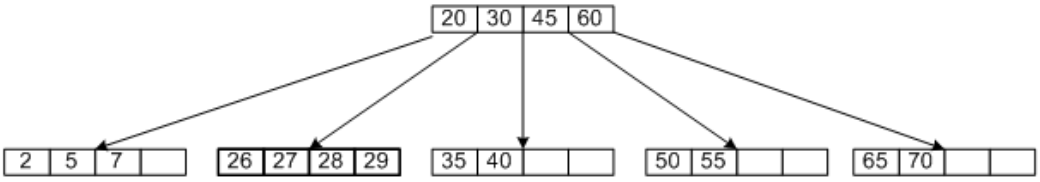


Рис. 5.16



25

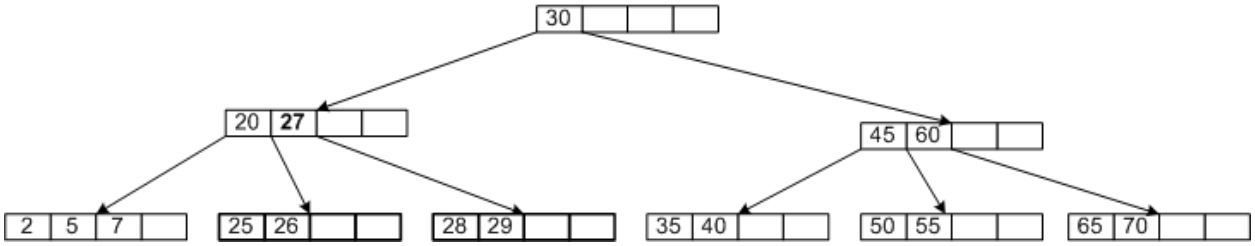


Рис. 5.17

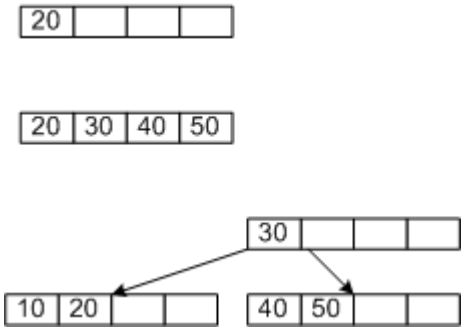


Рис. 5.18

5.5 Treaps

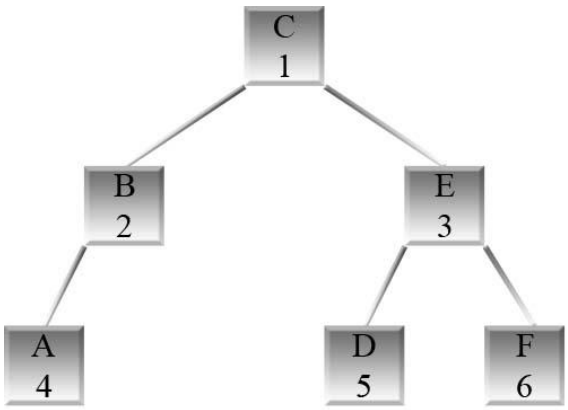


Рис. 5.19 *Treap*. Верх каждой вершины – её ключ, низ – приоритет

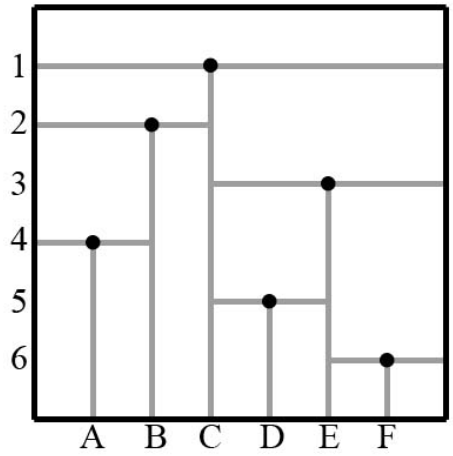
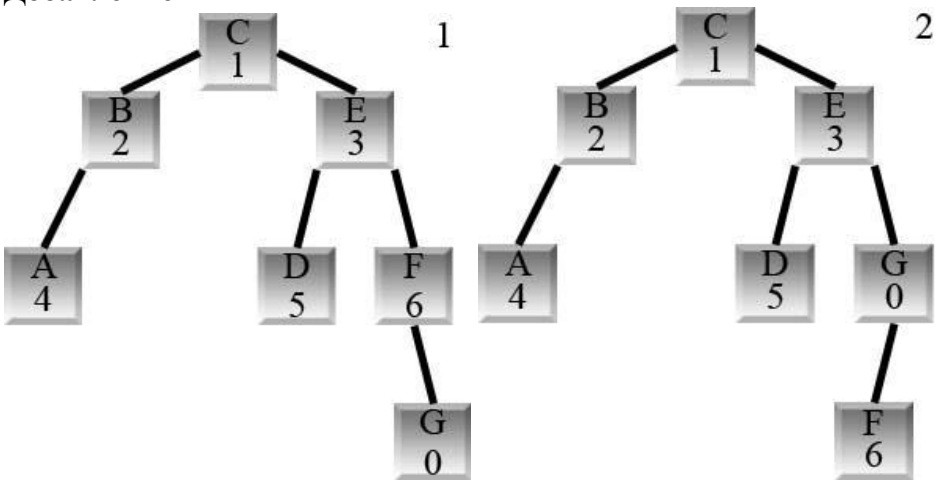


Рис. 5.20 Геометрическая интерпретация *treap*, показанного на рис. 5.19.

Добавление



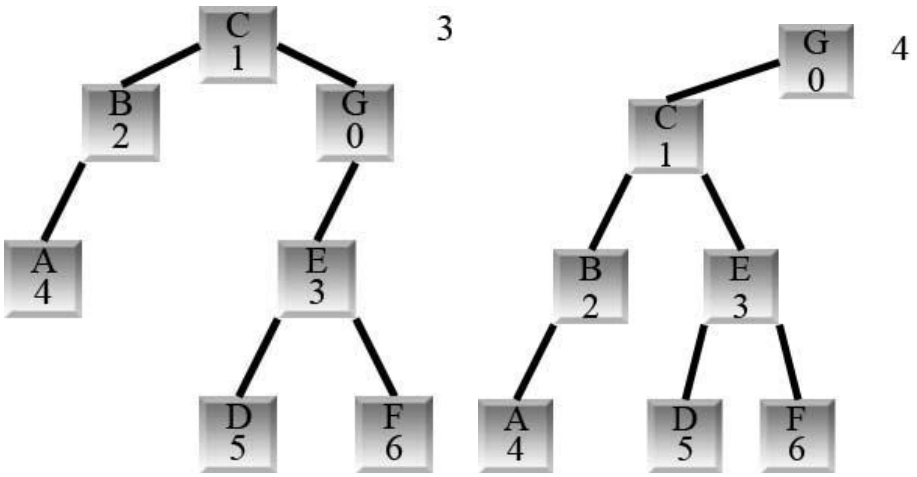
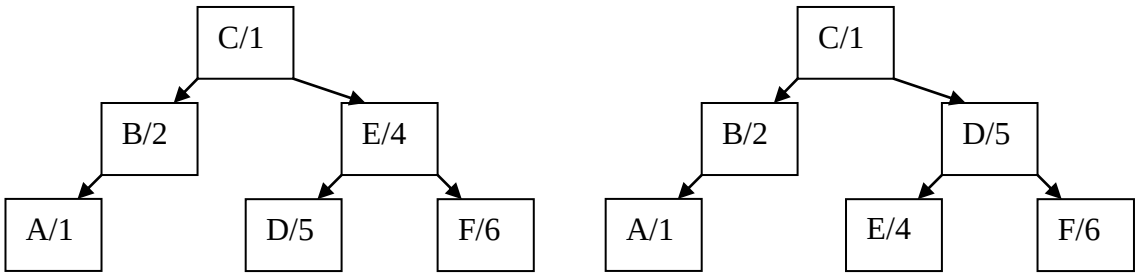


Рис. 5.21

Случайные приоритеты => H=2*ln(n)

Удаление

G/0 – см картинки в обратном порядке. Когда станет листом - удаляем.
Другой пример – удалить E:



Расцепление

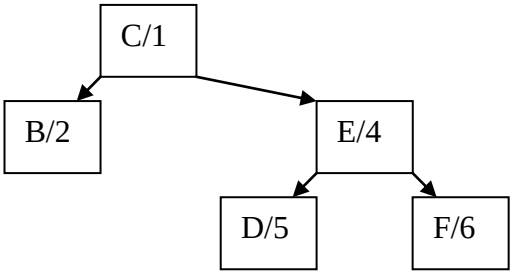
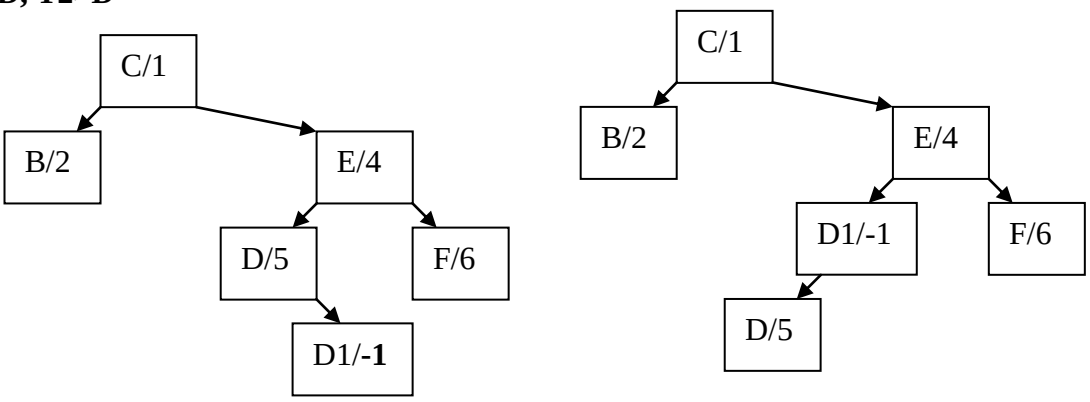


Рис. 4.

T1<=D, T2>D



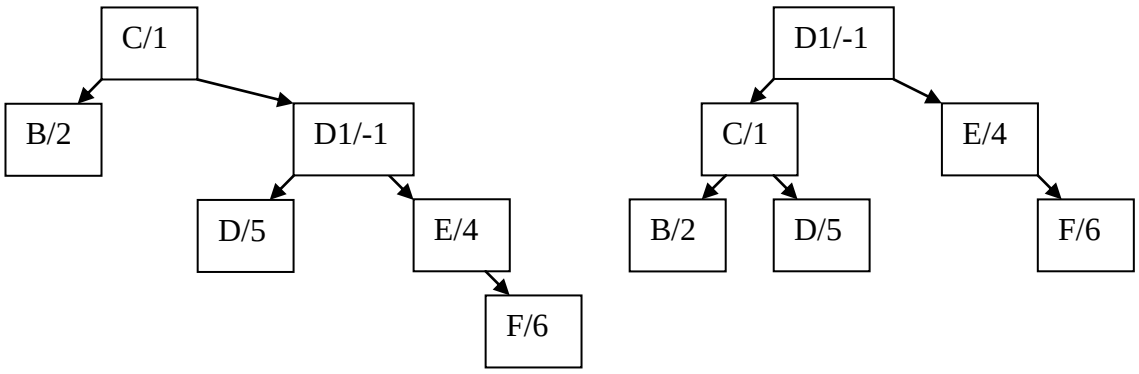


Рис. 5.

Сцепление

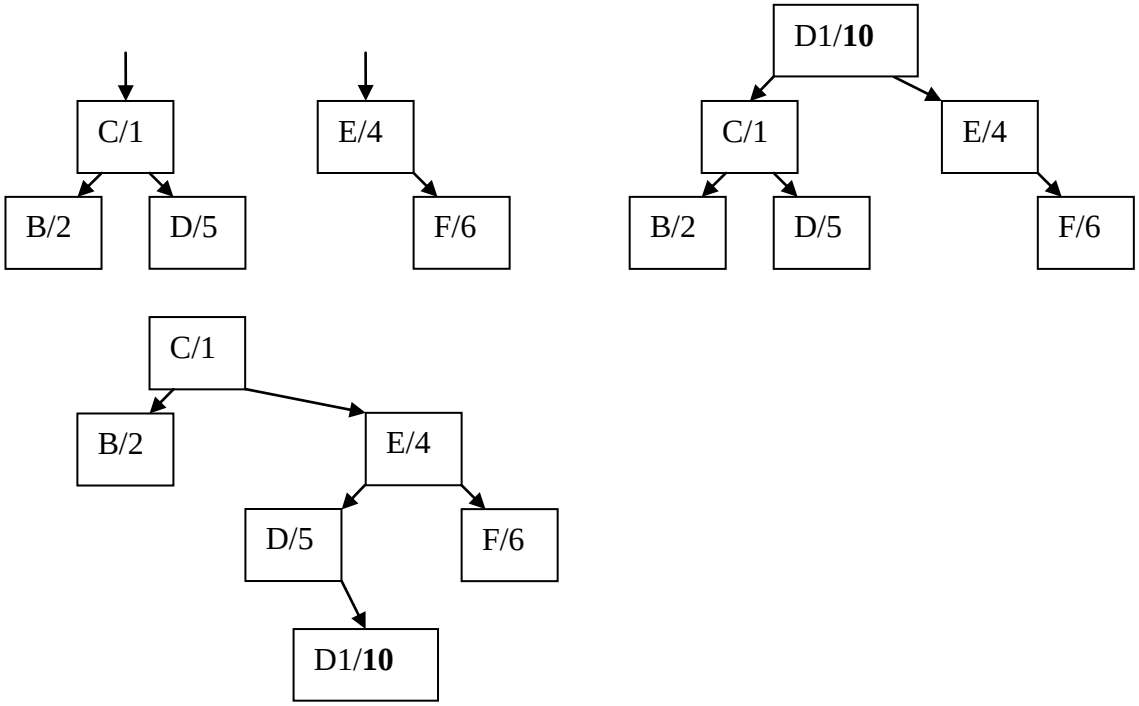


Рис. 6.

Сцепляемые очереди.
То же для деревьев двоичного поиска.

5.6 Скошенные деревья

Это самоорганизующиеся деревья двоичного поиска.
Очевидные подходы:
1 – рис 7 один поворот
2 - рис 7 продолжать пока X не станет корнем
Много лучше следующая эвристика.
После поиска X если он не корень дерева, выполняем следующий шаг:

Case 1: Если родитель X — корень дерева, то совершаем один поворот, делая X корнем дерева.

Рис. 7.

Case 2: Если P(X) не корень дерева, а X и P(X) оба левые или оба правые дети, делаем два поворота

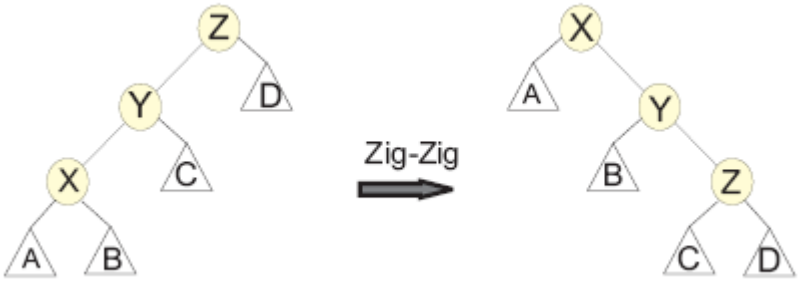


Рис. 8.

(можно только 1 поворот – повернуть только ребро Y-Z)

Case 3: Если P(X) не корень дерева и X— левый ребенок, а P(X) — правый ребенок, или наоборот, делаем два поворота

Рис. 9.

Амортизационная сложность - O((n+m)log n), где m- число операций в случайной последовательности.

Сцепляемые очереди

5.7 2-3 дерево

Определение и поиск

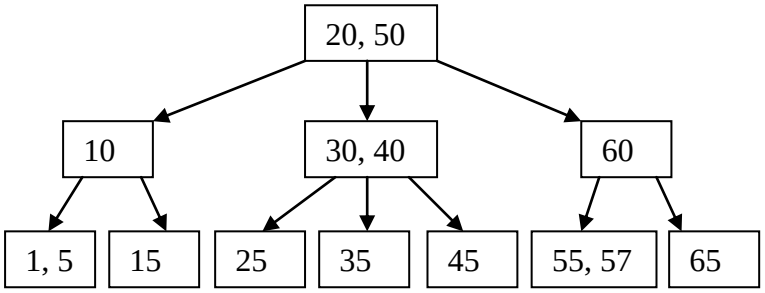


Рис. 10

$\log_3 n <= h <= \log_2 n$

Добавление

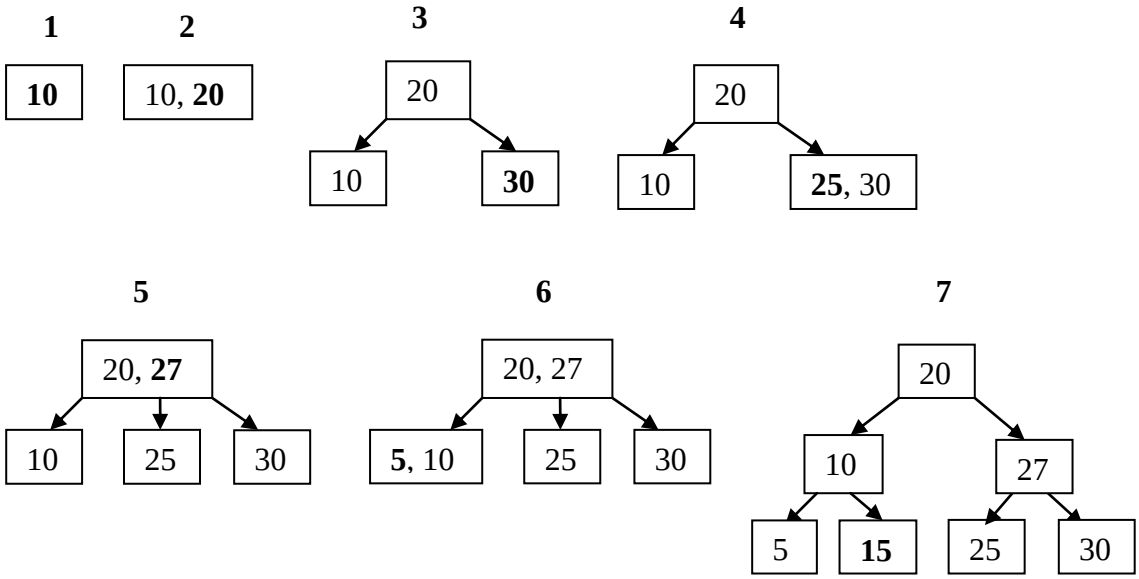


Рис. 11

Не обязательно до корня:

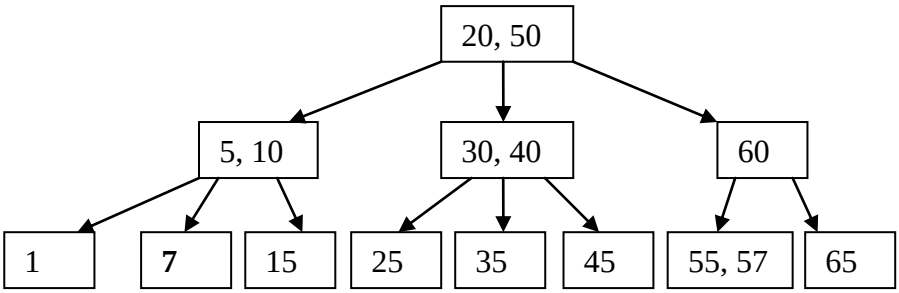


Рис. 12

Удаление

2-3+ дерево

Определение и поиск

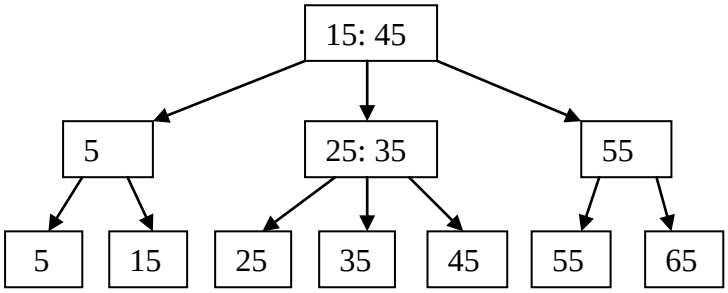


Рис. 13

Добавление

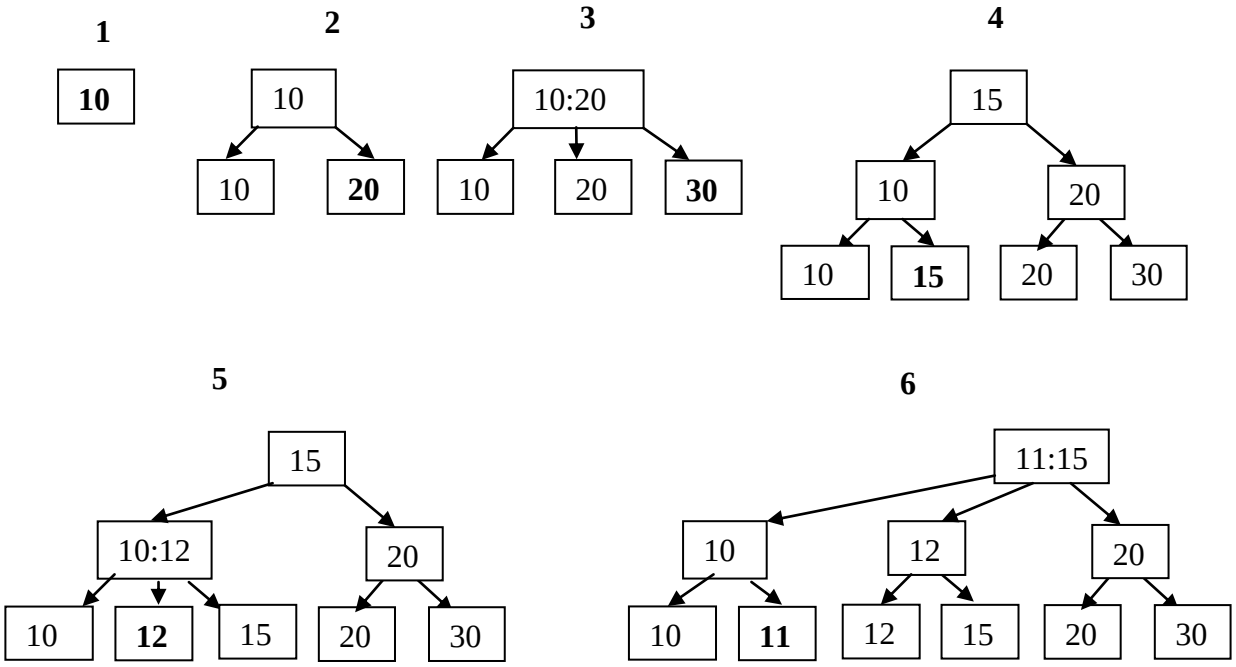


Рис. 14

Сравнение по памяти
Два сына
Без + (2указателя +1ключ+1на данные)N = 4N
С+ (2указателя +1ключ)N+(1ключ+1на данные)N=5N
Три сына
Без + (3указателя +2ключа+1на данные)N = 6N
С+ (3указателя +2ключа)N/2+(1ключ+1на данные)N=4,5N

Удаление

Случай 1

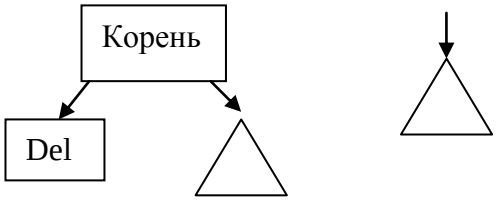


Рис. 15

Случай 2

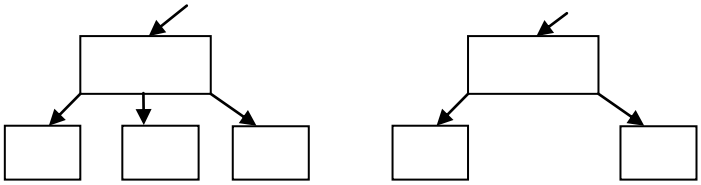


Рис. 16

Случай 3

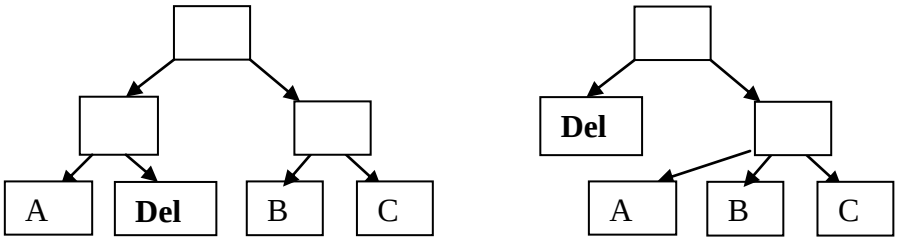


Рис. 17

Случай 4

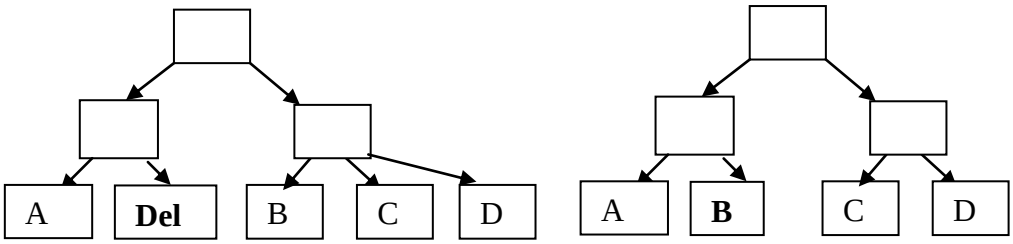


Рис. 18

Сцепление

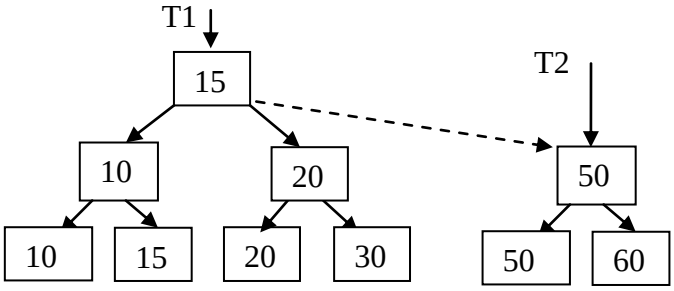


Рис. 19

Расцепление

Расцепление по a

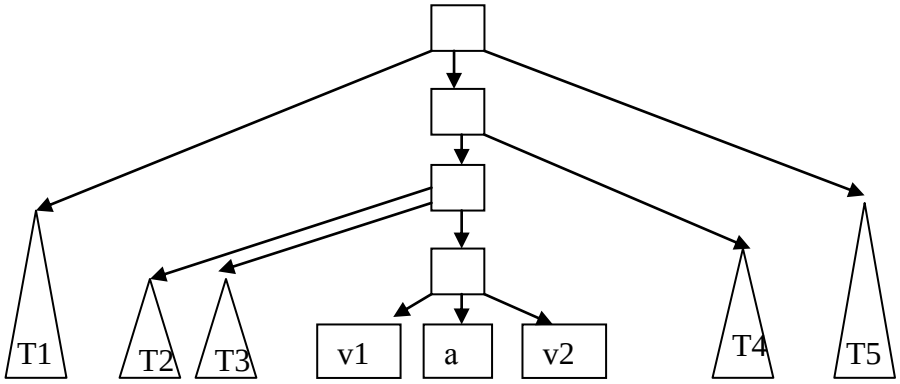


Рис. 20

T1, T2, T3, v1, a – сцепить
v2, T4, T5 – сцепить
Обработка всех

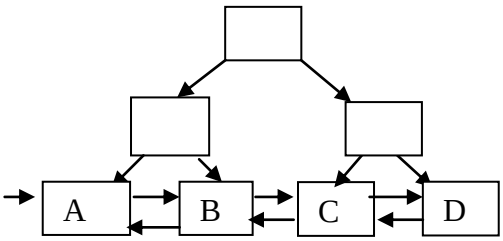
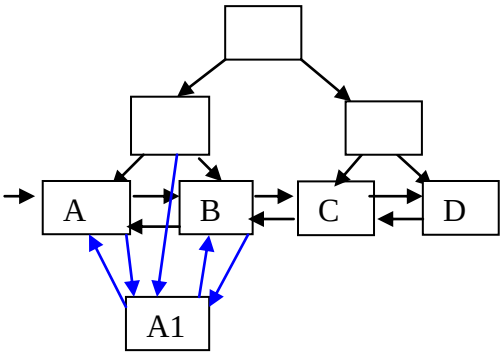
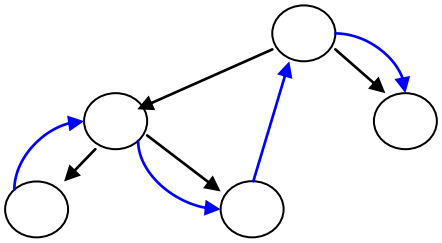
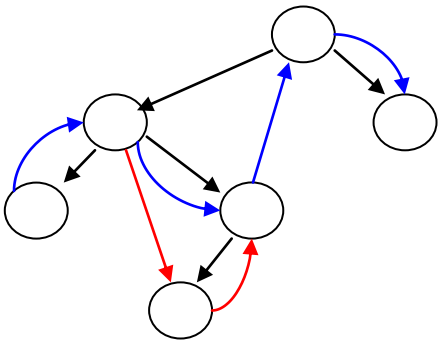
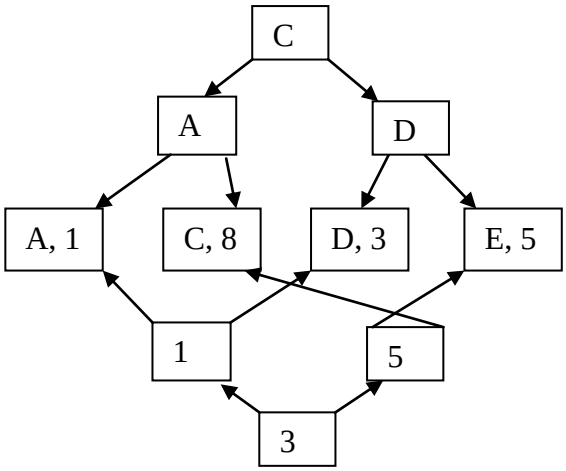


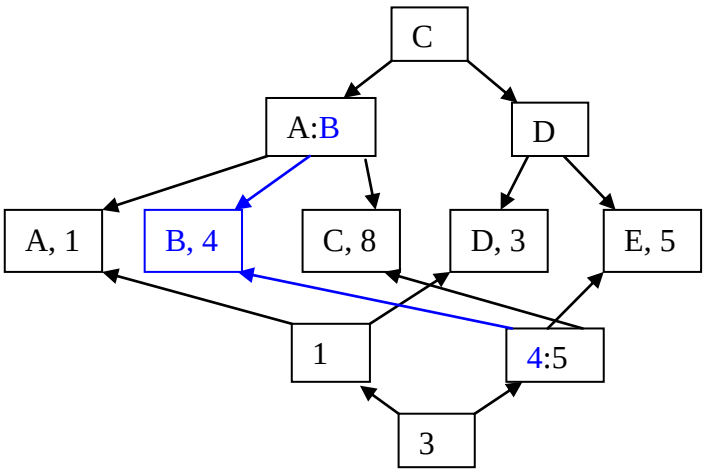
Рис. 21











5.8 B – деревья

Определение и поиск

Min – t-1

Max – 2t-1

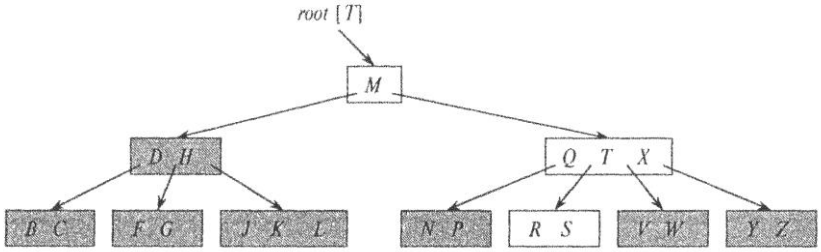


Рис. 22 t=3

Добавление

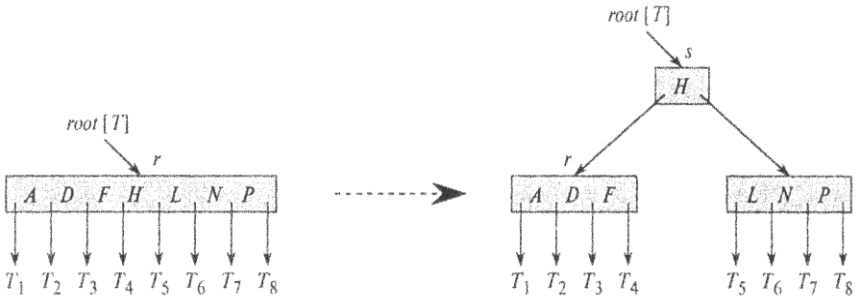


Рис. 23 t=4 Разбиение корня

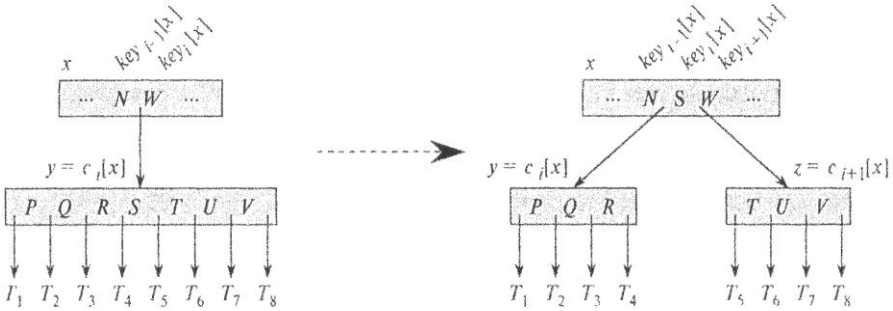


Рис. 24 t=4 Разбиение узла

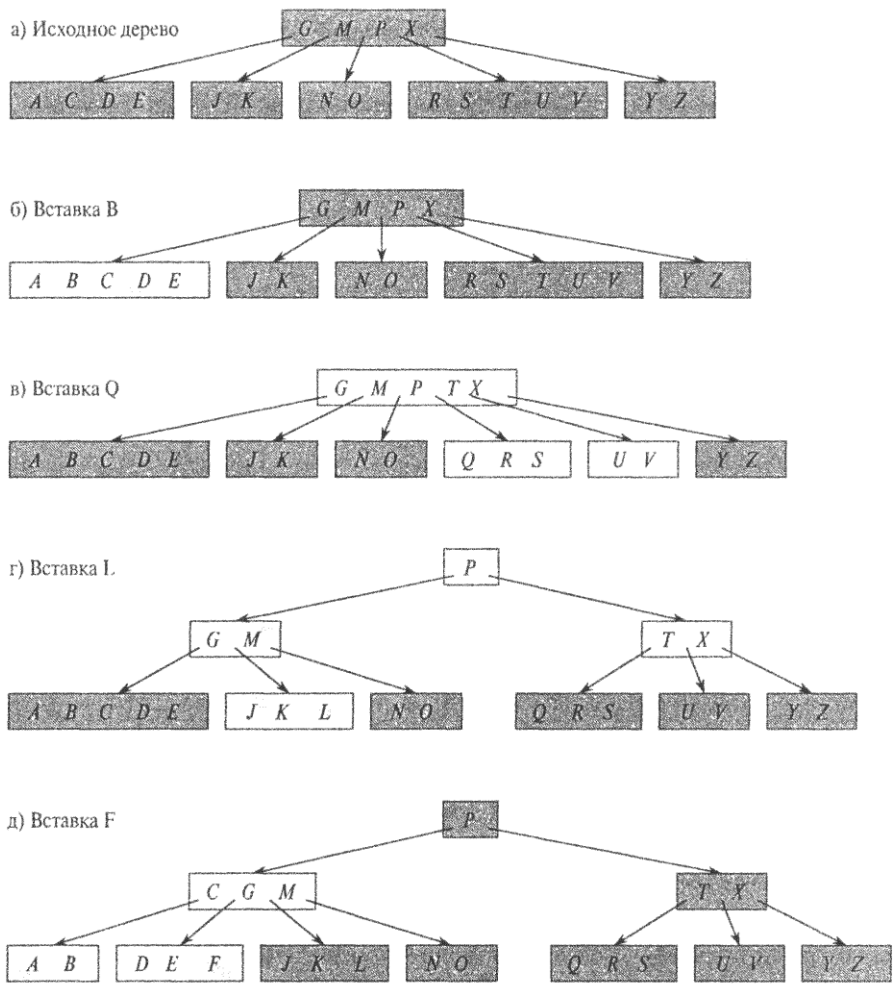


Рис. 25 t=3

B+ деревья

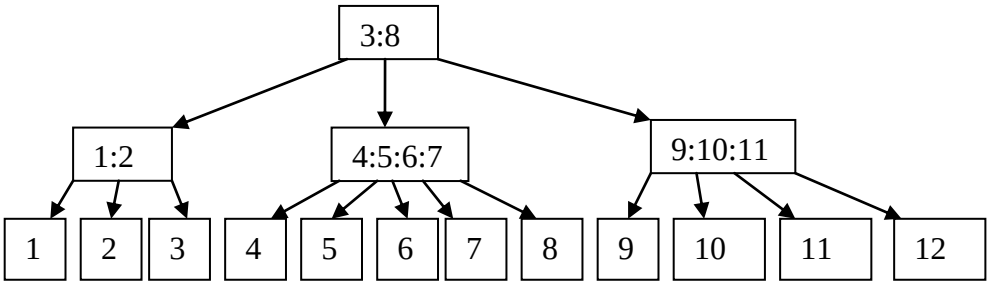


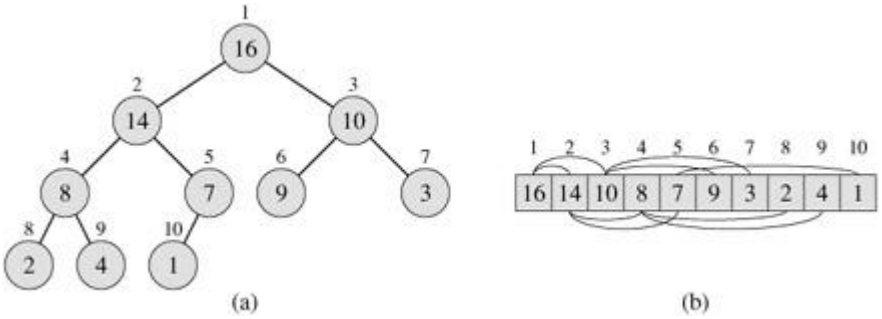
Рис. 26 t=3

Преимущества B+

6 Кучи

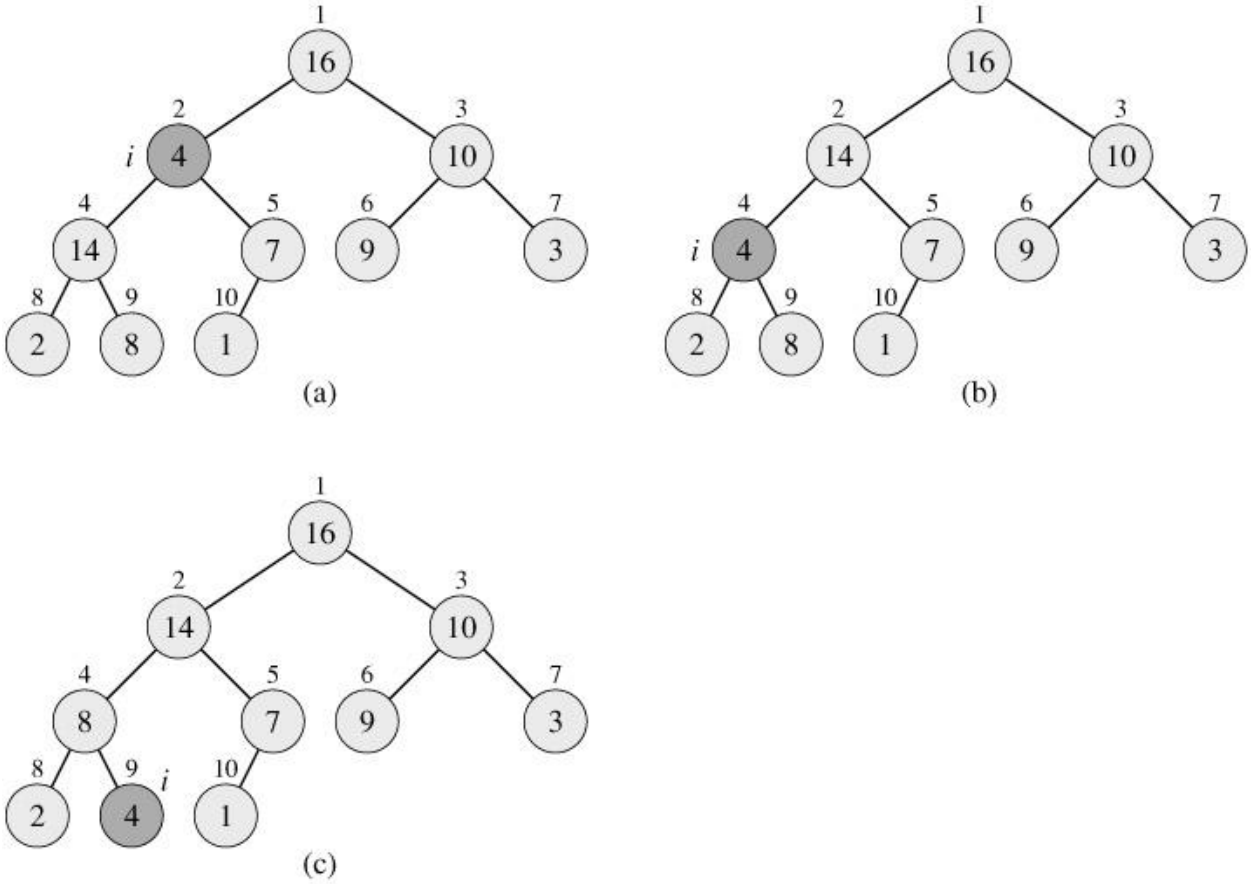
6.1 Кучи на массиве

6.1.1 Определение




```
PARENT(i)
   return ⌊i/2⌋

LEFT(i)
   return 2i

RIGHT(i)
   return 2i + 1
```

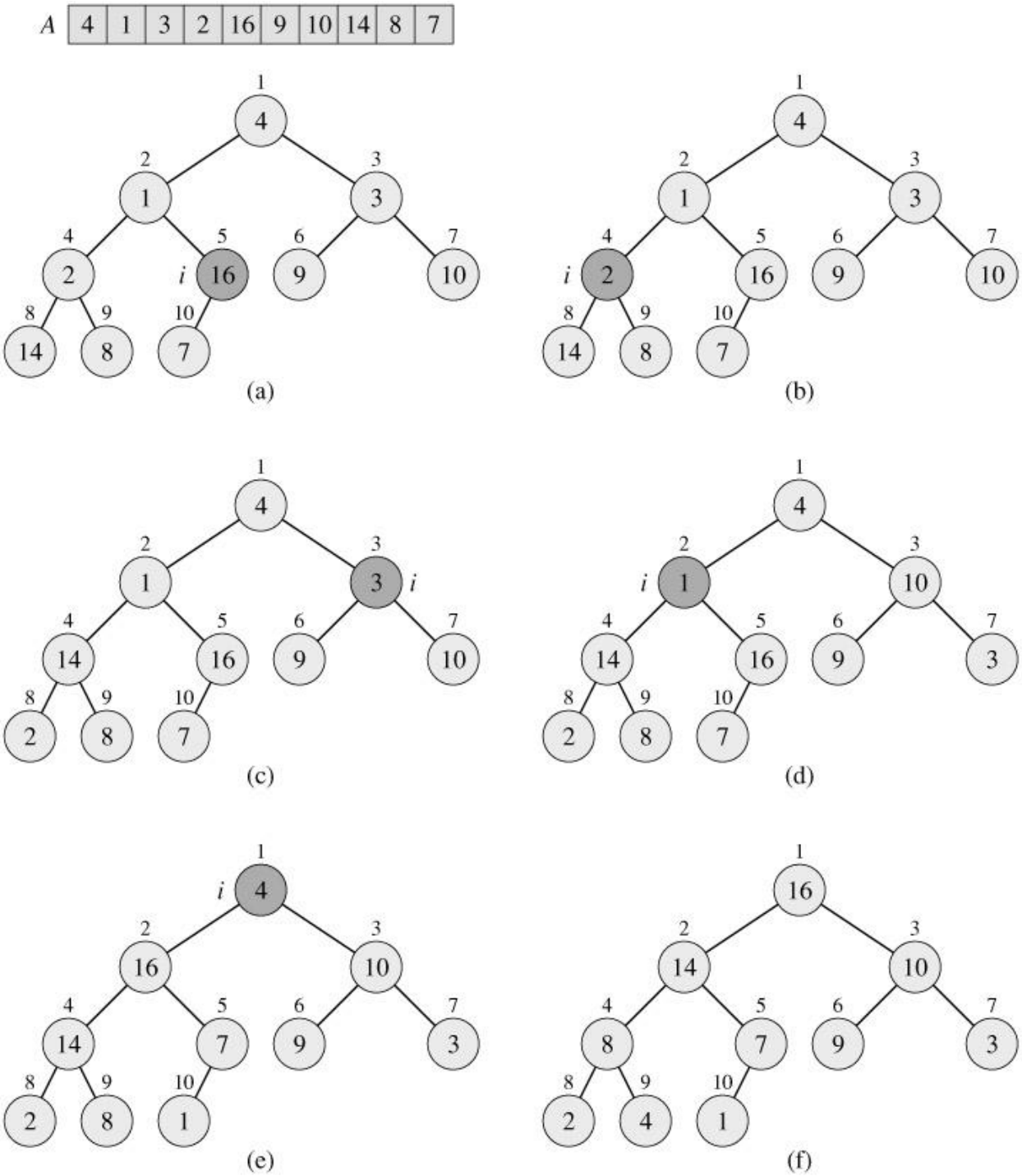
6.1.2 Поддержка свойств кучи




```
MAX-HEAPIFY(A, i)
```

```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ heap-size[A] and A[l] > A[i]
   then largest ← l
   else largest ← i
if r ≤ heap-size[A] and A[r] > A[largest]
   then largest ← r
if largest ≠ i
   then exchange A[i] ↔ A[largest]
        MAX-HEAPIFY(A, largest)
```

6.1.3 Построение кучи



```
BUILD-MAX-HEAP(A)
1  heap-size[A] ← length[A]
2  for i ← ⌊length[A]/2⌋ downto 1
3       do MAX-HEAPIFY(A, i)
```

H=lg n; Число узлов на высоте h (считая снизу) = 2^{H-h}
Сложность MAX-HEAPIFY(A, i) = h

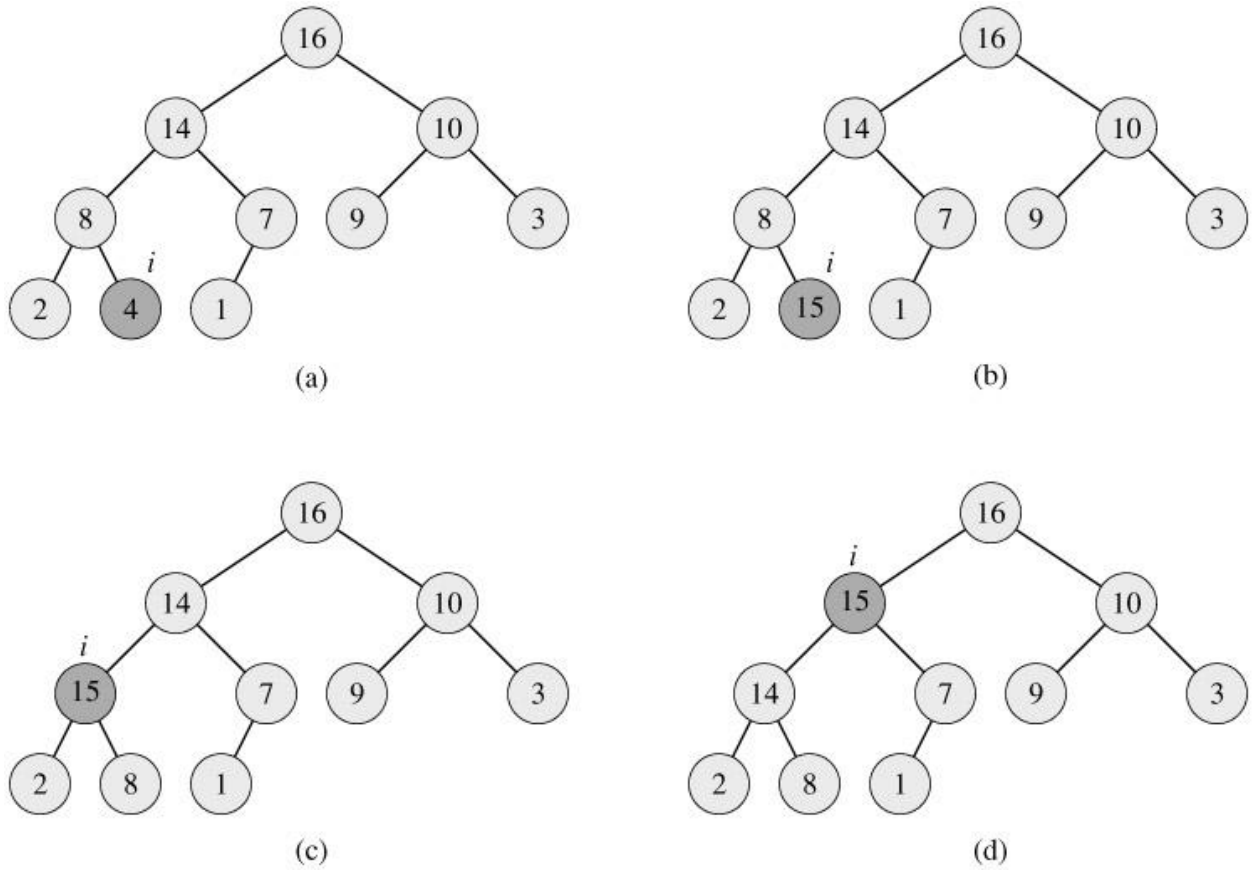
$$T(n) = \sum_{h=1}^{\lg n} \frac{n}{2^h} O(h) = nO(\sum_{h=1}^{\lg n} \frac{h}{2^h}) < nO(\sum_{h=1}^{\infty} \frac{h}{2^h}) = Cn$$

6.1.4 Очередь с приоритетами

INSERT(S, x), MAXIMUM(S), EXTRACT-MAX(S), INCREASE-KEY(S, x, k)

```
HEAP-EXTRACT-MAX(A)
1 if heap-size[A] < 1
2   then error "heap underflow"
3 max ← A[1]
4 A[1] ← A[heap-size[A]]
5 heap-size[A] ← heap-size[A] - 1
6 MAX-HEAPIFY(A, 1)
7 return max
```

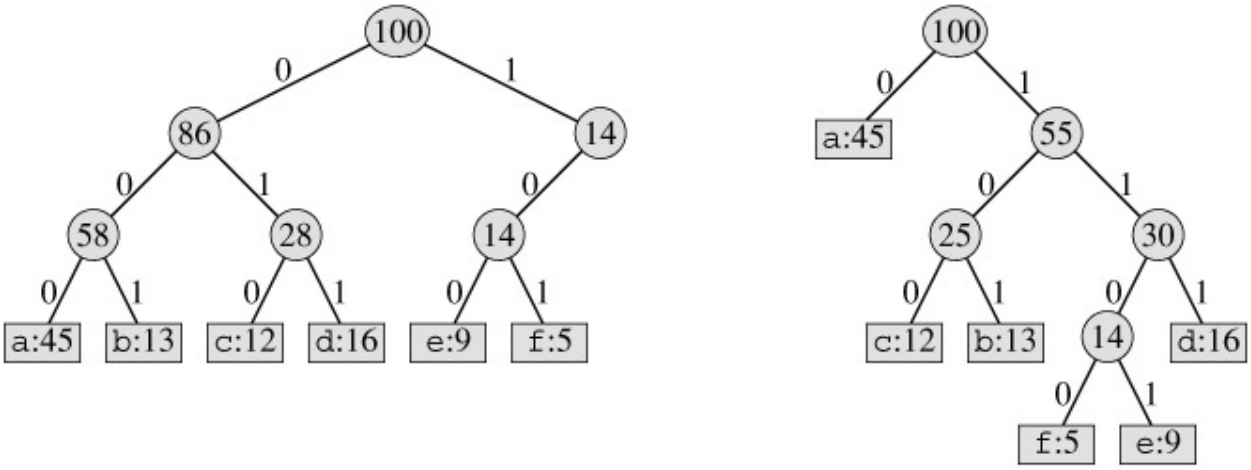
```
HEAP-INCREASE-KEY(A, i, key)
1 if key < A[i]
2   then error "new key is smaller than current key"
3 A[i] ← key
4 while i > 1 and A[PARENT(i)] < A[i]
5     do exchange A[i] ↔ A[PARENT(i)]
6         i ← PARENT(i)
```




```
MAX-HEAP-INSERT(A, key)
1 heap-size[A] ← heap-size[A] + 1
2 A[heap-size[A]] ← -∞
3 HEAP-INCREASE-KEY(A, heap-size[A], key)
```

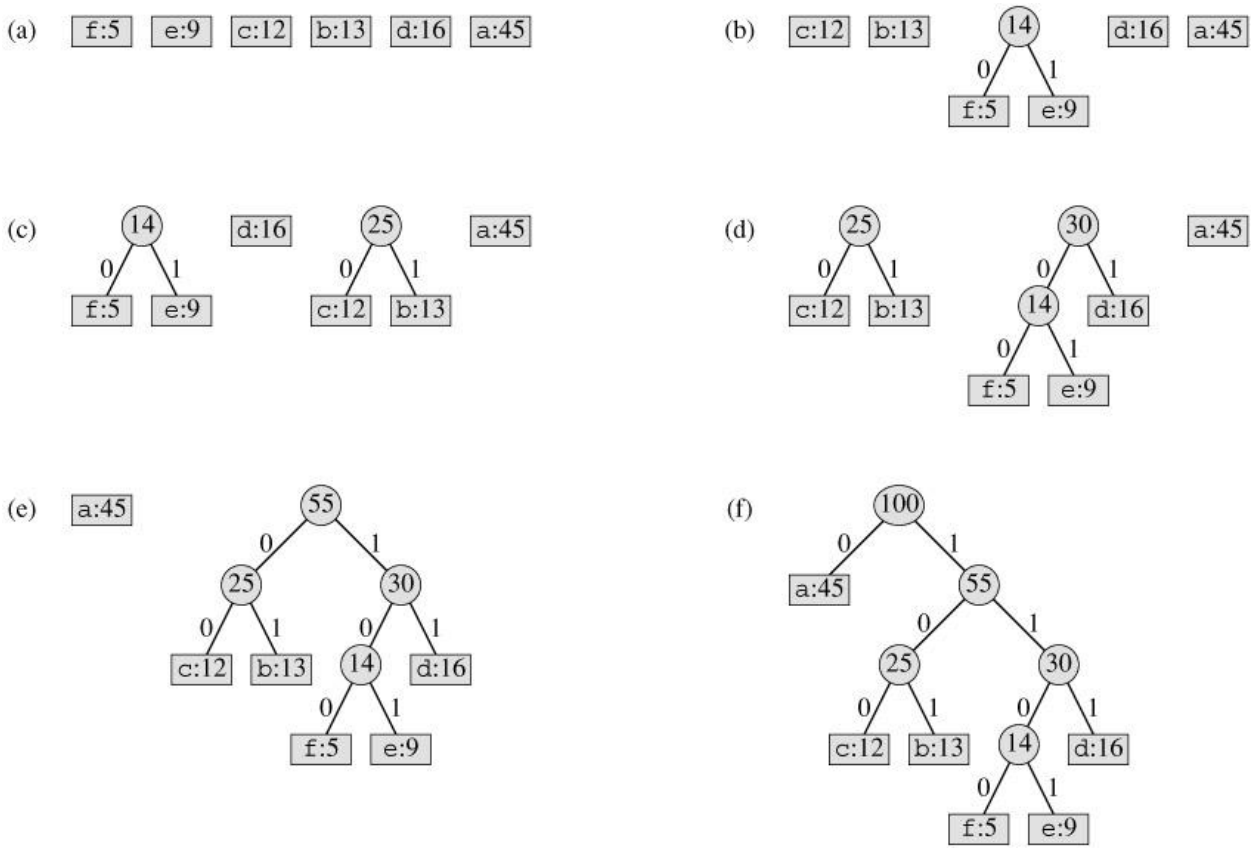
6.1.5 Huffman коды

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

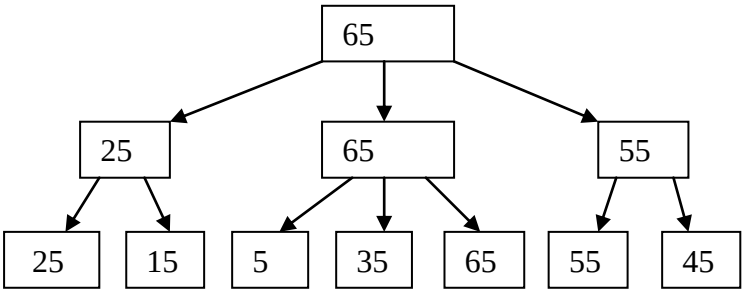


(a) (b)

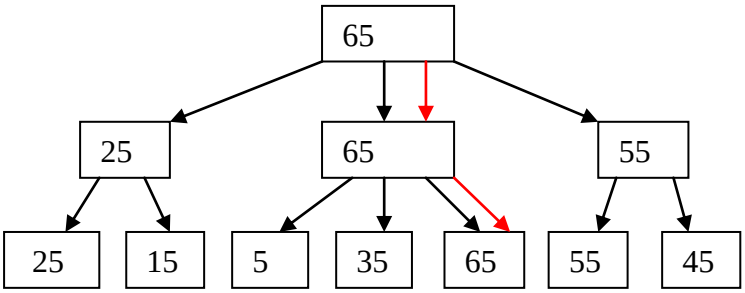
```
HUFFMAN(C)
n ← |C|
Q ← C
for i 1 to n - 1
     do allocate a new node z
        left[z] ← x ← EXTRACT-MIN (Q)
        right[z] ← y ← EXTRACT-MIN (Q)
        f [z] ← f [x] + f [y]
        INSERT(Q, z)
return EXTRACT-MIN(Q)   ▹Return the root of the tree
```



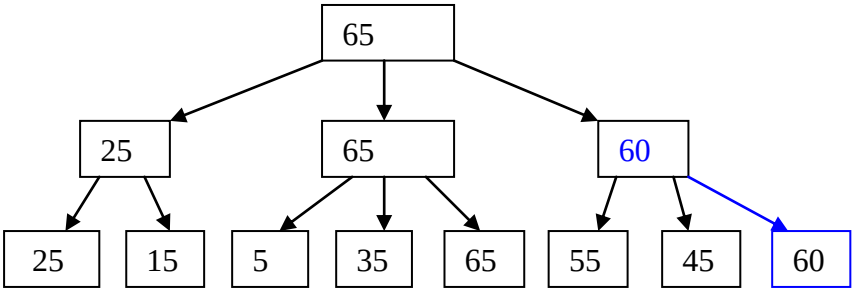
6.2 Кучи на 2-3+ деревьях



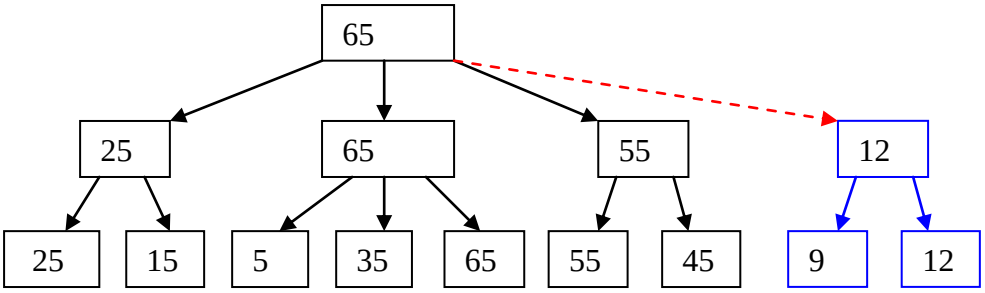
Посмотреть максимальный или забрать его

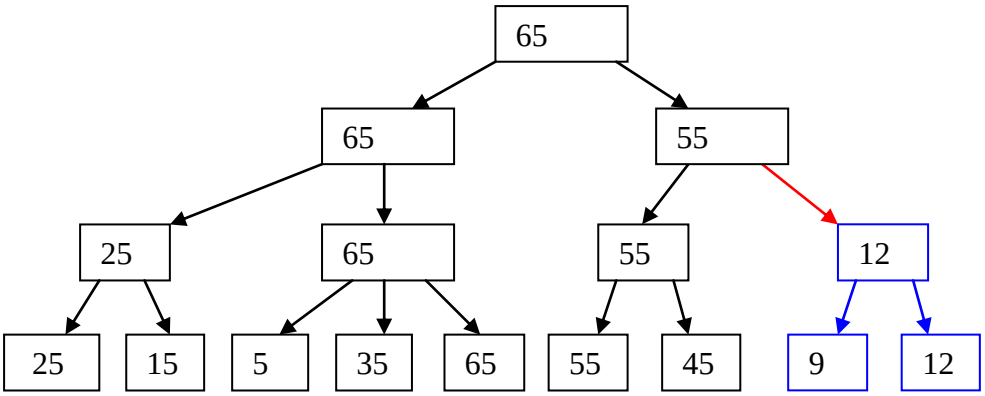


Добавить 60



Объединить





6.3 Биномиальные кучи

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Binomial trees

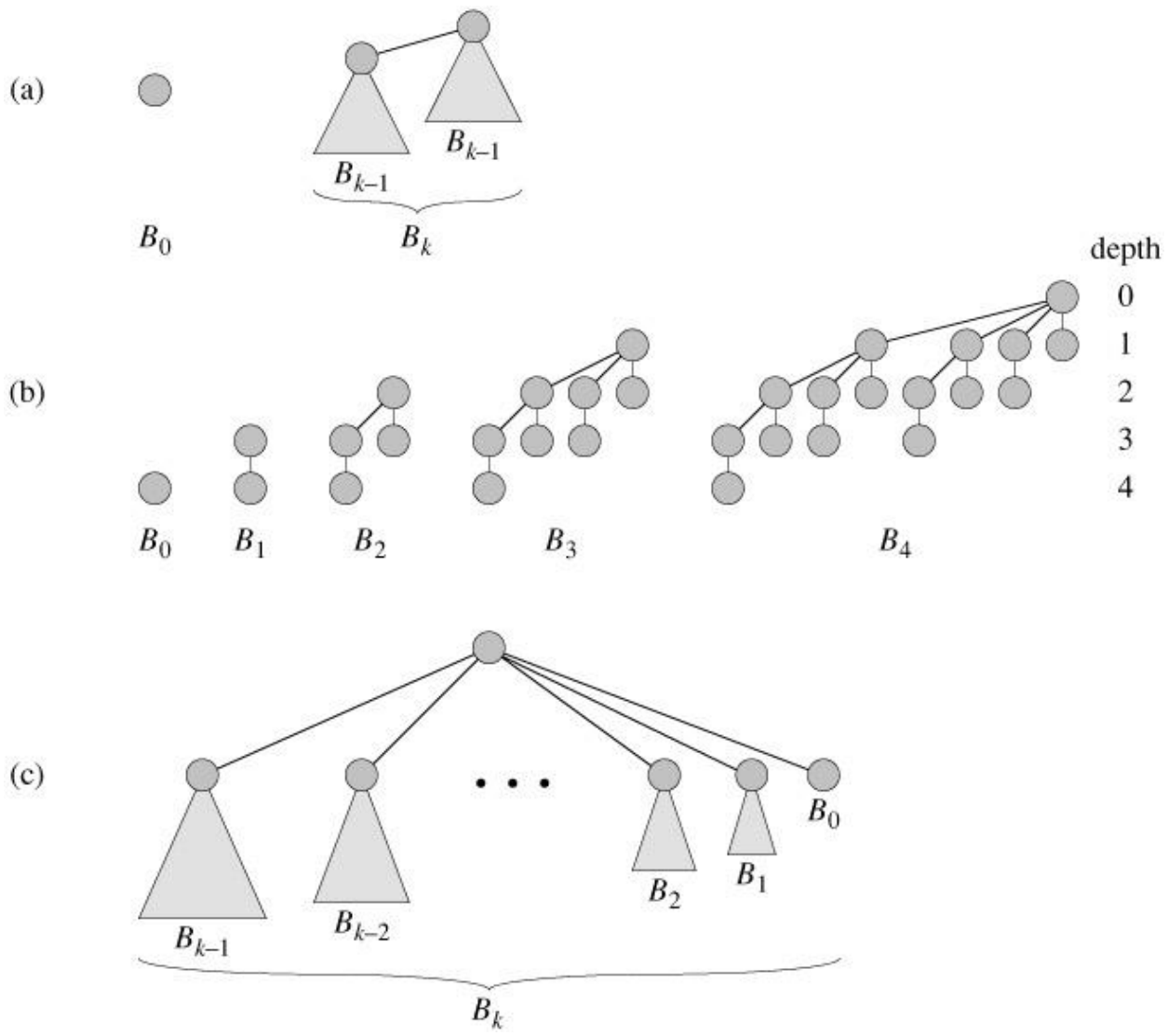


Рис. 1
Binomial heaps

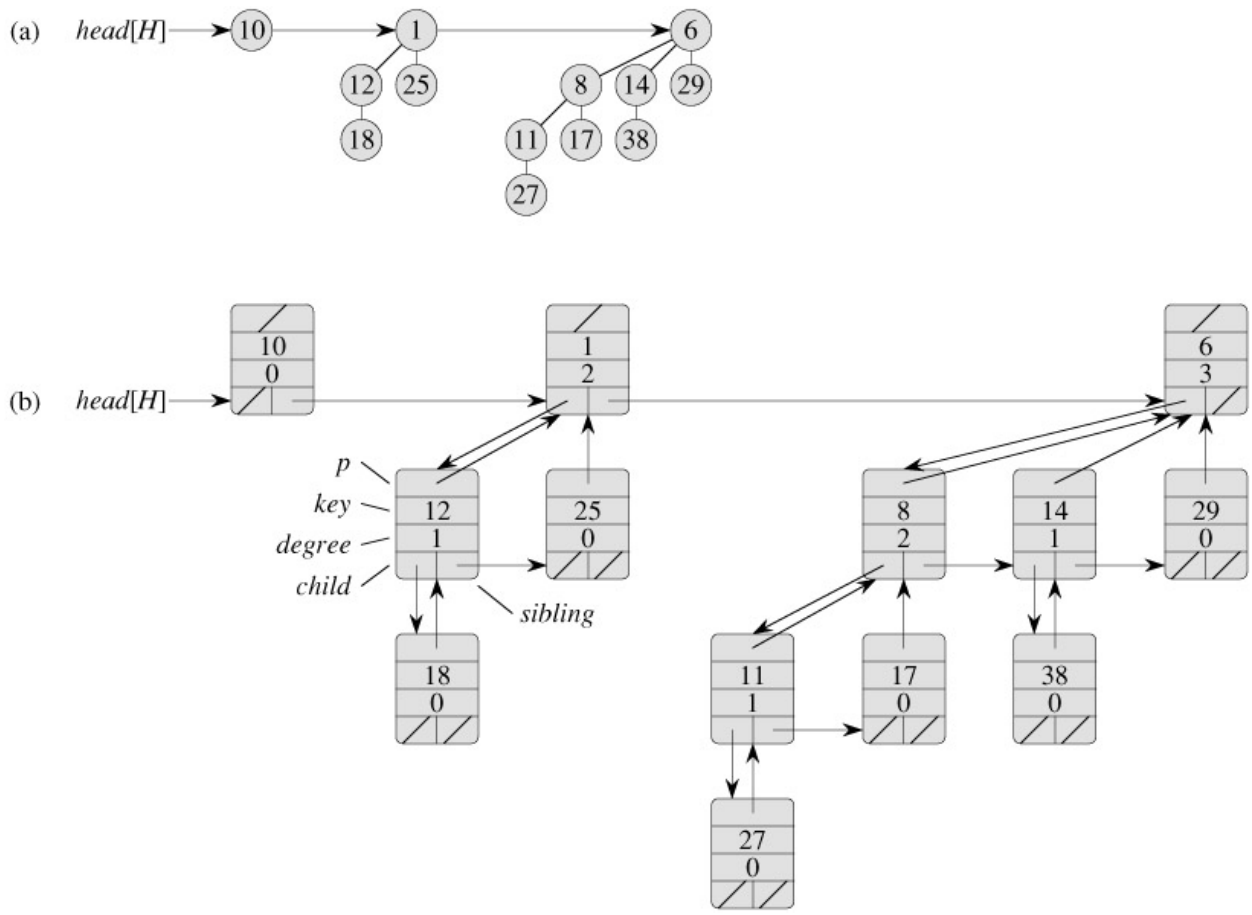


Рис. 2

Finding the minimum key

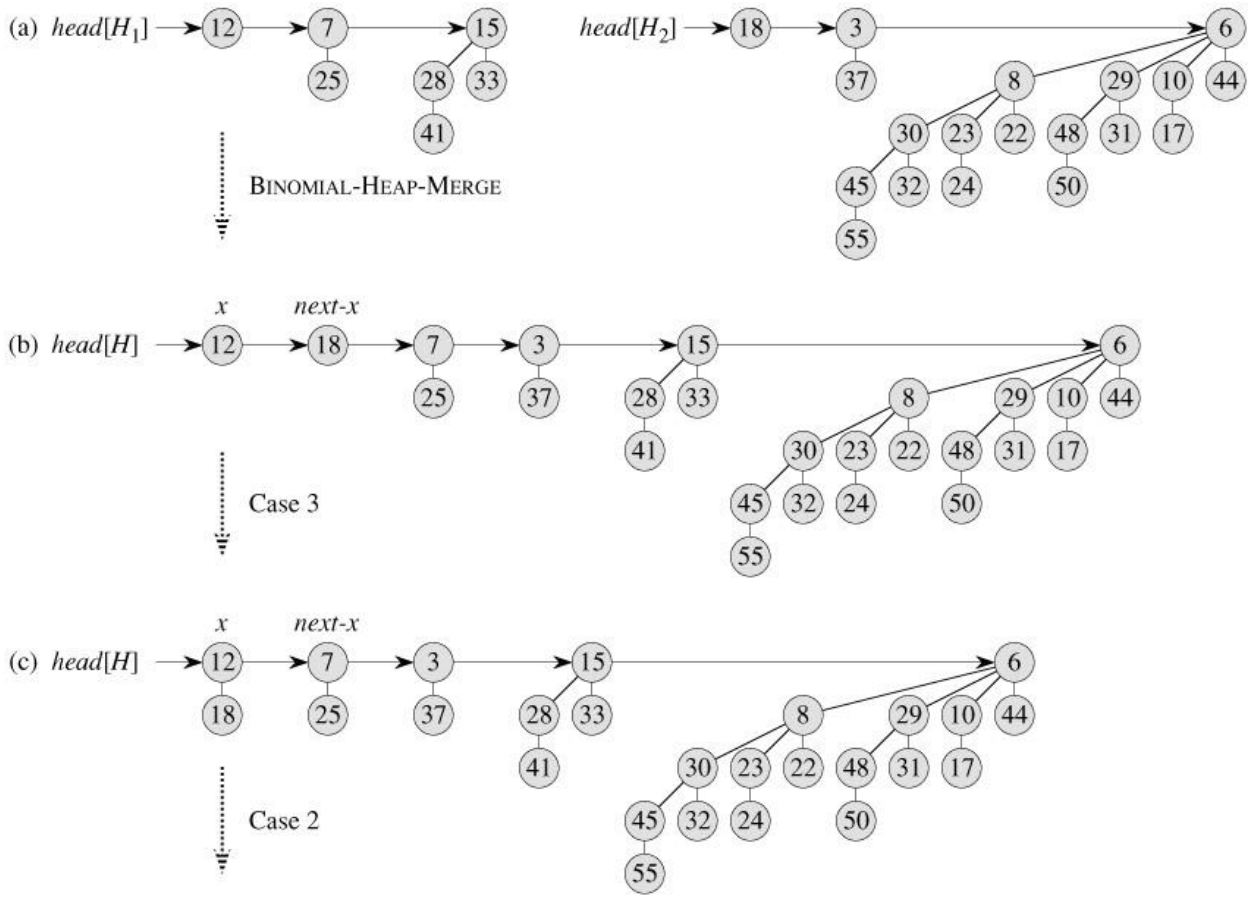
```
BINOMIAL-HEAP-MINIMUM(H)
y ← NIL
x ← head[H]
min ← ∞
while x ≠ NIL
   do if key[x] < min
         then min ← key[x]
              y ← x
      x ← sibling[x]
return y
```

Uniting two binomial heaps

```
BINOMIAL-LINK(y, z)
p[y] ← z
sibling[y] ← child[z]
child[z] ← y
degree[z] ← degree[z] + 1
```

```
BINOMIAL-HEAP-UNION(H1, H2)
H ← MAKE-BINOMIAL-HEAP()
head[H] ← BINOMIAL-HEAP-MERGE(H1, H2)
```

```
free the objects H1 and H2 but not the lists they point to
if head[H] = NIL
   then return H
prev-x ← NIL
x ← head[H]
next-x ← sibling[x]
while next-x ≠ NIL
    do if (degree[x] ≠ degree[next-x]) or
          (sibling[next-x] ≠ NIL and degree[sibling[next-x]] = degree[x])
          then prev-x ← x                                  ▷ Cases 1 and 2
               x ← next-x                                  ▷ Cases 1 and 2
          else if key[x] ≤ key[next-x]
                  then sibling[x] ← sibling[next-x]          ▷ Case 3
                       BINOMIAL-LINK(next-x, x)               ▷ Case 3
                  else if prev-x = NIL                        ▷ Case 4
                          then head[H] ← next-x              ▷ Case 4
                          else sibling[prev-x] ← next-x       ▷ Case 4
                       BINOMIAL-LINK(x, next-x)               ▷ Case 4
                       x ← next-x                            ▷ Case 4
       next-x ← sibling[x]
return H
```



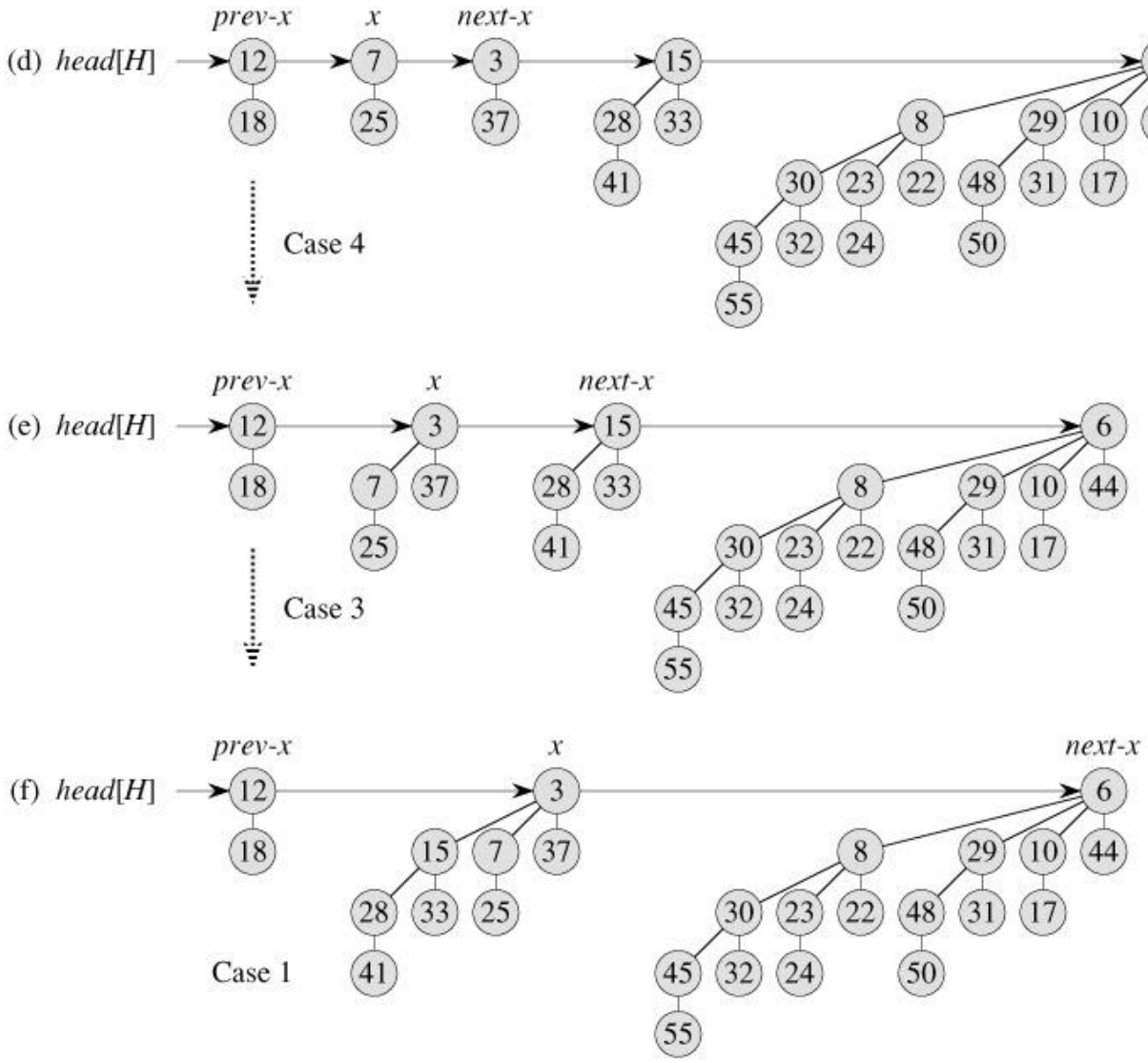

(d) head[H]
prev-x
x
next-x
Case 4
(e) head[H]
prev-x
x
next-x
Case 3
(f) head[H]
prev-x
x
next-x
Case 1

Рис. 3

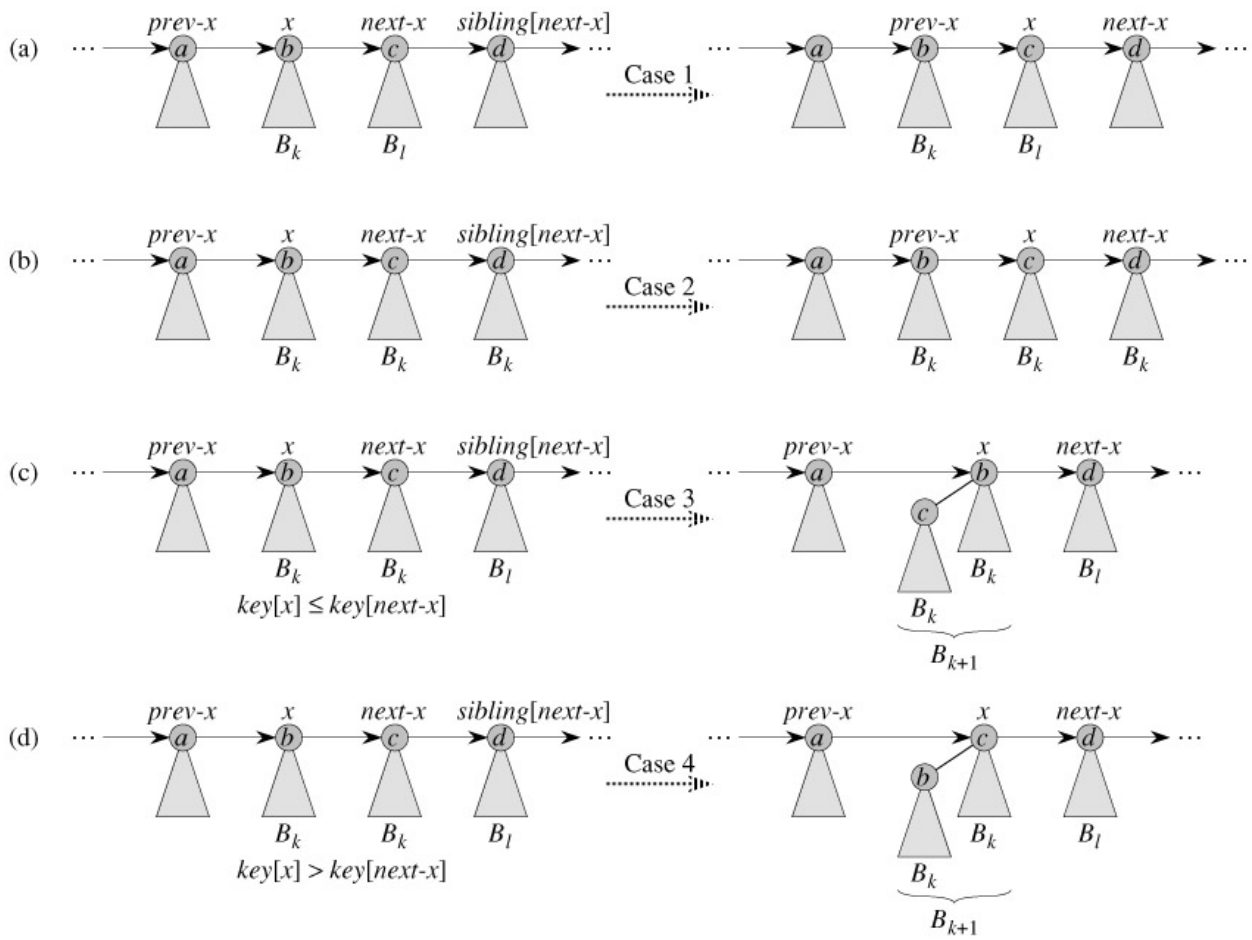


Рис. 4

1.3.4.1 Inserting a node

```
BINOMIAL-HEAP-INSERT(H, x)
1  H' ← MAKE-BINOMIAL-HEAP()
2  p[x] ← NIL
3  child[x] ← NIL
4  sibling[x] ← NIL
5  degree[x] ← 0
6  head[H'] ← x
7  H ← BINOMIAL-HEAP-UNION(H, H')
```

1.3.4.2 Extracting the node with minimum key

```
BINOMIAL-HEAP-EXTRACT-MIN(H)
1  find the root x with the minimum key in the root list of H,
           and remove x from the root list of H
2  H' ← MAKE-BINOMIAL-HEAP()
3  reverse the order of the linked list of x's children,
           and set head[H'] to point to the head of the resulting list
4  H ← BINOMIAL-HEAP-UNION(H, H')
5  return x
```

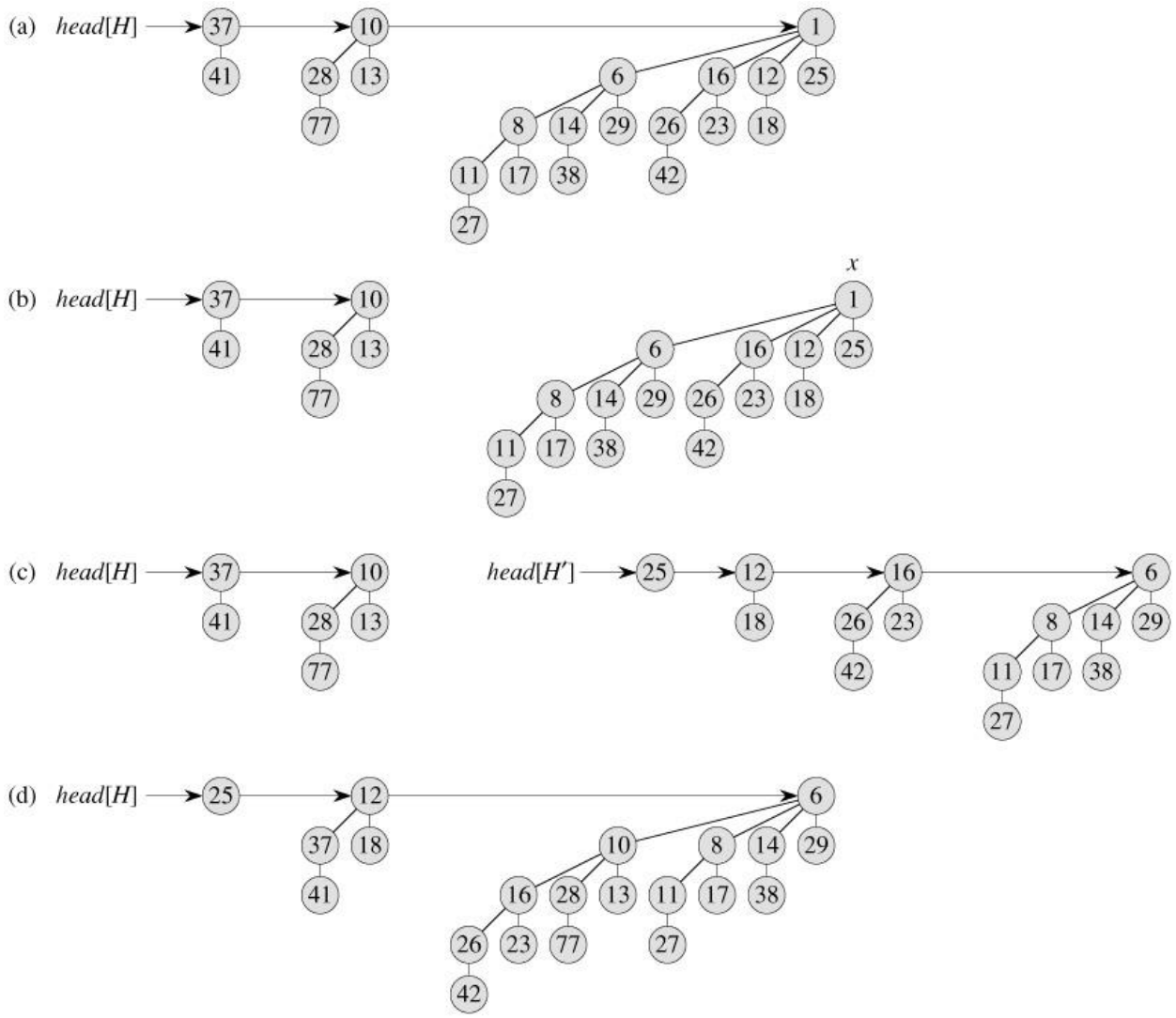


Рис. 5

1.3.4.3 Decreasing a key

```
BINOMIAL-HEAP-DECREASE-KEY(H, x, k)
if k > key[x]
   then error "new key is greater than current key"
key[x] ← k
y ← x
z ← p[y]
while z ≠ NIL and key[y] < key[z]
    do exchange key[y] ↔ key[z]
       ▸ If y and z have satellite fields, exchange them, too.
       y ← z
       z ← p[y]
```

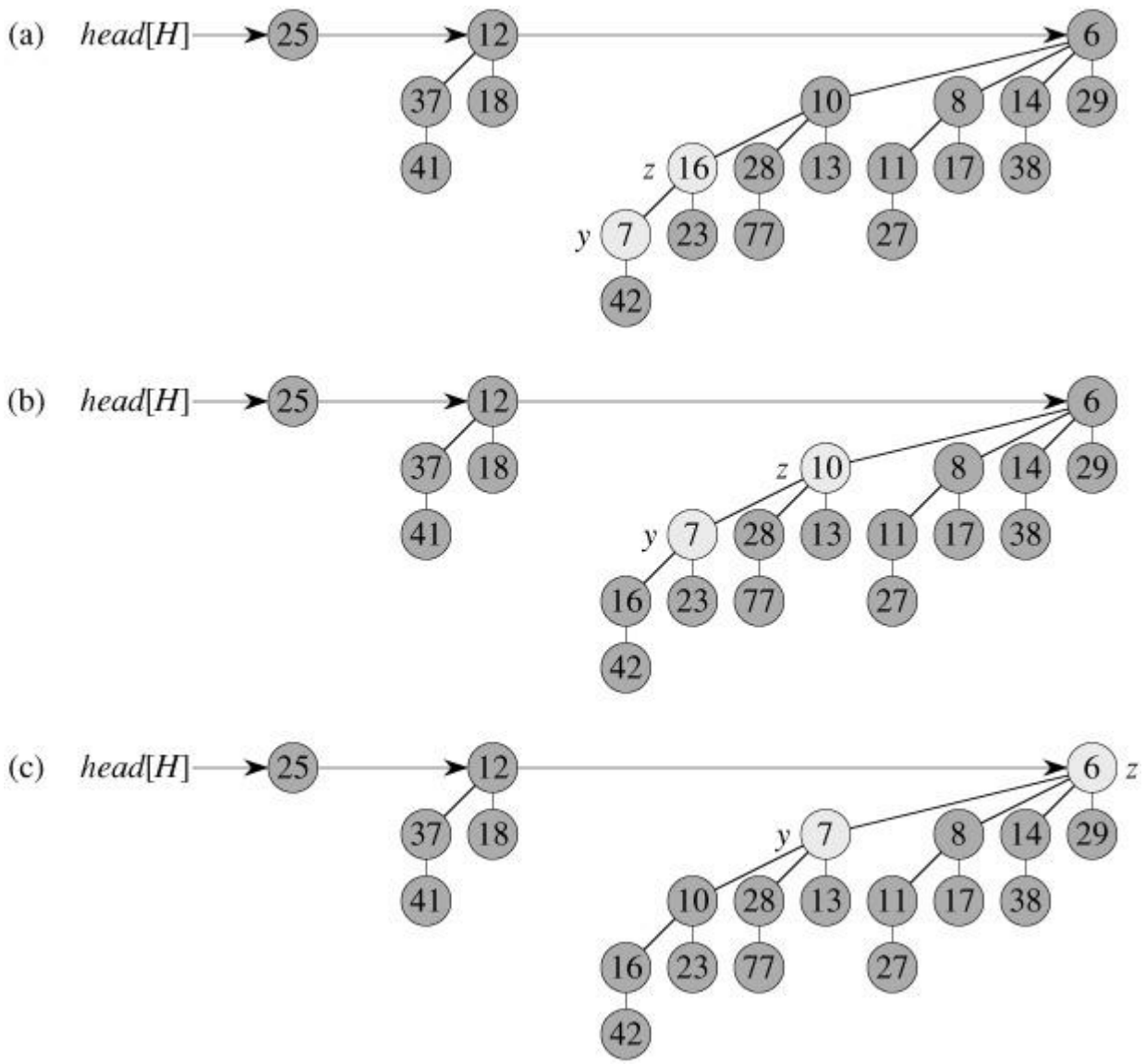


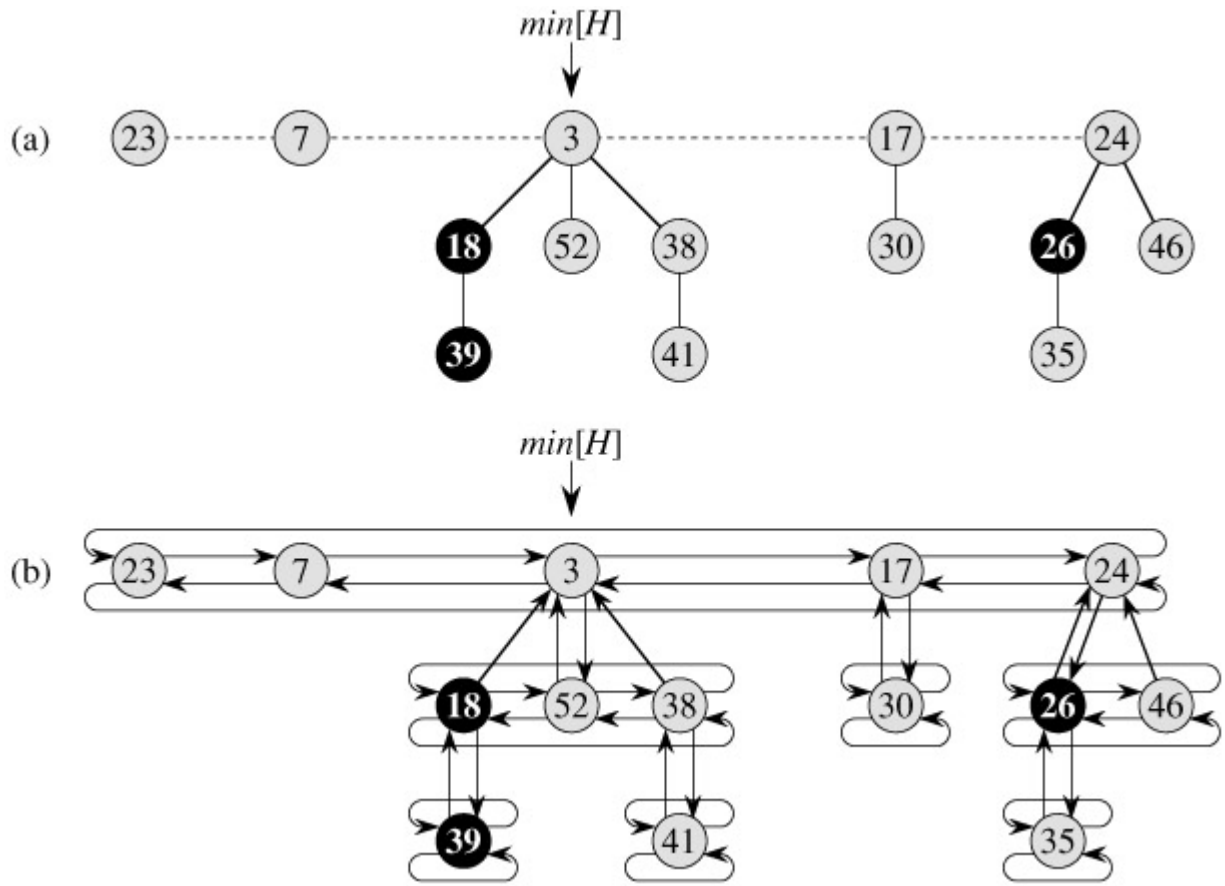
Рис. 6

1.3.4.4 Deleting a key

```
BINOMIAL-HEAP-DELETE(H, x)
1  BINOMIAL-HEAP-DECREASE-KEY(H, x, -∞)
2  BINOMIAL-HEAP-EXTRACT-MIN(H)
```

6.4 Fibonacci Heaps

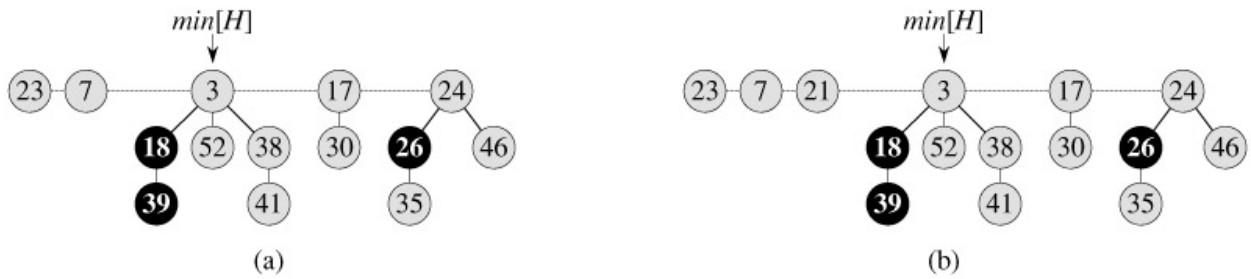
Structure of Fibonacci heaps



Pic. 1

1.3.4.5 Inserting a node

```
FIB-HEAP-INSERT(H, x)
 1  degree[x] ← 0
 2  p[x] ← NIL
 3  child[x] ← NIL
 4  left[x] ← x
 5  right[x] ← x
 6  mark[x] ← FALSE
 7  concatenate the root list containing x with root list H
 8  if min[H] = NIL or key[x] < key[min[H]]
 9     then min[H] ← x
10  n[H] ← n[H] + 1
```



Pic. 2

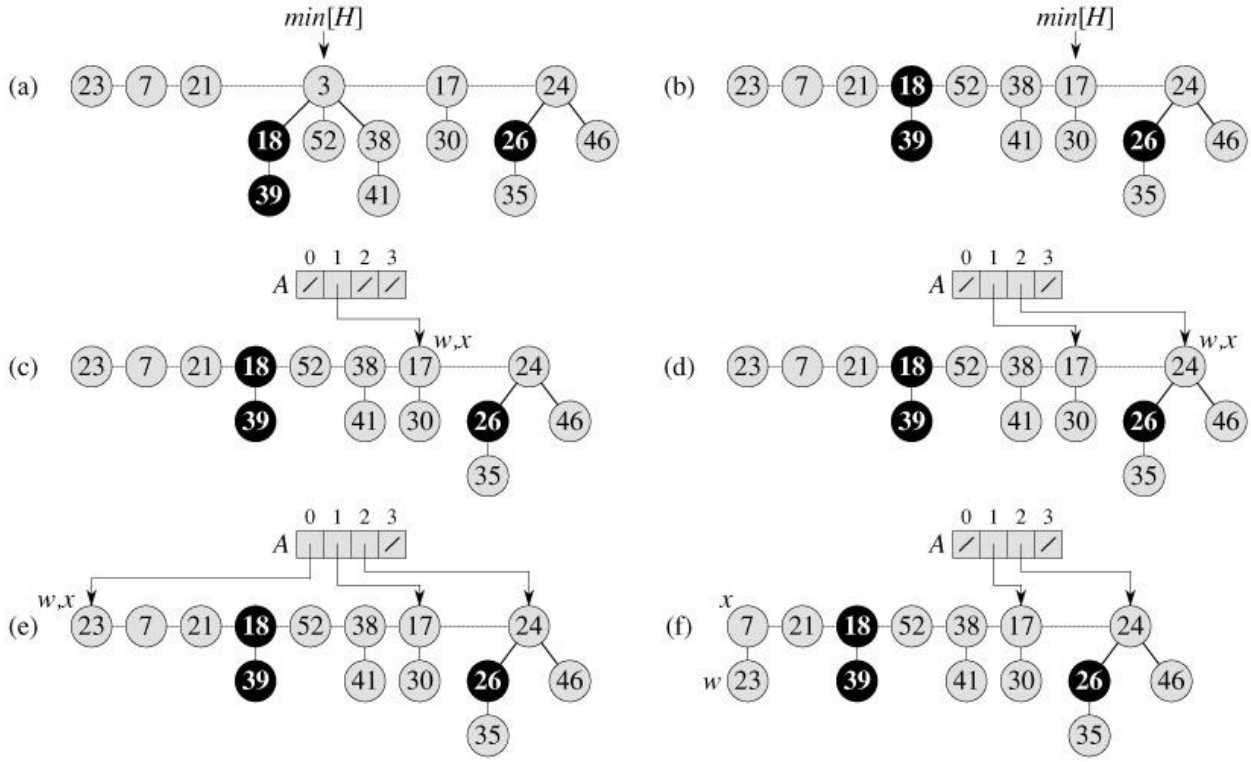
1.3.4.6 Uniting two Fibonacci heaps

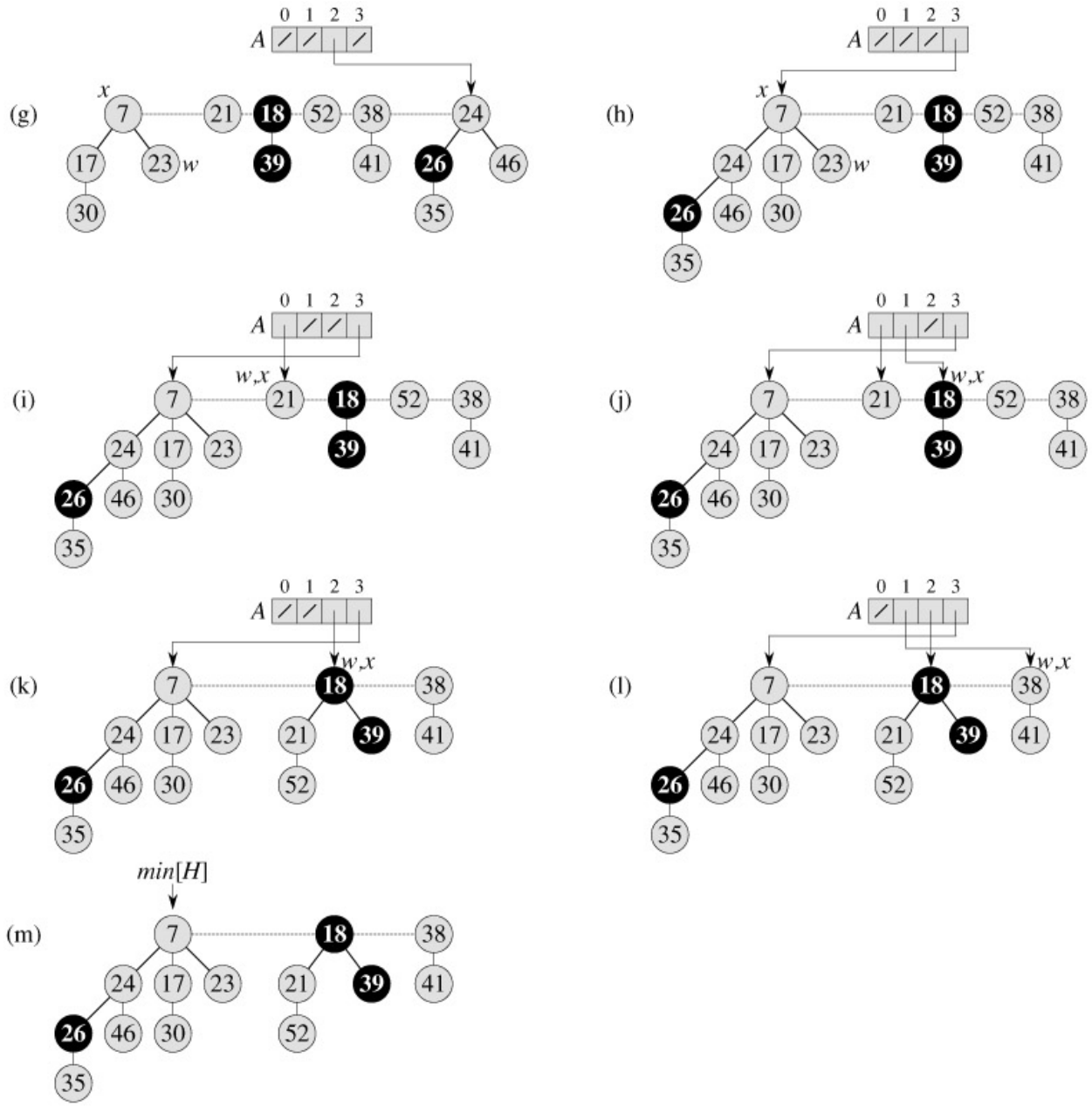
```
FIB-HEAP-UNION(H1, H2)
```

```
H ← MAKE-FIB-HEAP()
min[H] ← min[H1]
concatenate the root list of H2 with the root list of H
if (min[H1] = NIL) or (min[H2] ≠ NIL and min[H2] < min[H1])
  then min[H] ← min[H2]
n[H] ← n[H1] + n[H2]
free the objects H1 and H2
return H
```

1.3.4.7 Extracting the minimum node

```
FIB-HEAP-EXTRACT-MIN(H)
z ← min[H]
if z ≠ NIL
   then for each child x of z
            do add x to the root list of H
               p[x] ← NIL
        remove z from the root list of H
        if z = right[z]
           then min[H] ← NIL
           else min[H] ← right[z]
                CONSOLIDATE(H)
        n[H] ← n[H] - 1
return z
```





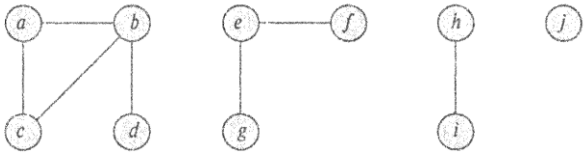
Pic. 3

```
CONSOLIDATE(H)
 1 for i ← 0 to D(n[H])
 2      do A[i] ← NIL
 3 for each node w in the root list of H
 4      do x ← w
 5         d ← degree[x]
 6         while A[d] ≠ NIL
 7             do y ← A[d]       ▷ Another node with the same degree as x.
 8                if key[x] > key[y]
 9                   then exchange x ↔ y
10                FIB-HEAP-LINK(H, y, x)
11                A[d] ← NIL
12                d ← d + 1
13         A[d] ← x
14 min[H] ← NIL
15 for i ← 0 to D(n[H])
16      do if A[i] ≠ NIL
17            then add A[i] to the root list of H
```

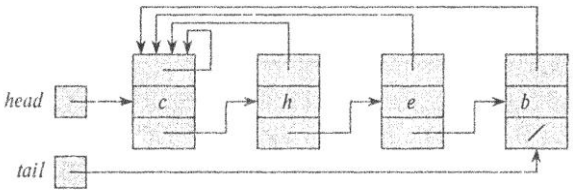
```
                    if min[H] = NIL or key[A[i]] < key[min[H]]
                       then min[H] ← A[i]
FIB-HEAP-LINK(H, y, x)
  remove y from the root list of H
  make y a child of x, incrementing degree[x]
  mark[y] ← FALSE
```

7 Структуры данных для непересекающихся множеств

Make_Set(x) x - представитель
Union(x, y)
Find_Set(x)

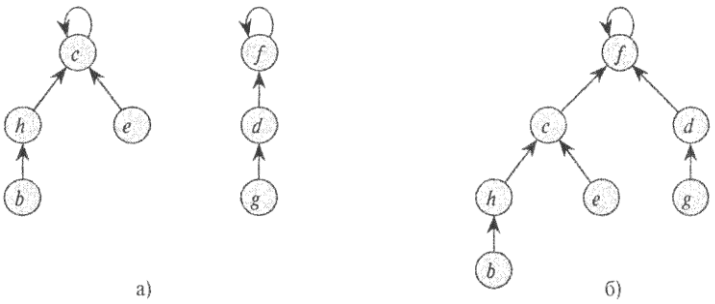


CONNECTED_COMPONENTS(G)
for Каждая вершина $v \in V[G]$
  do MAKE_SET(v)
for Каждое ребро $(u, v) \in E[G]$
  do if FIND_SET(u) $\neq$ FIND_SET(v)
    then UNION(u, v)

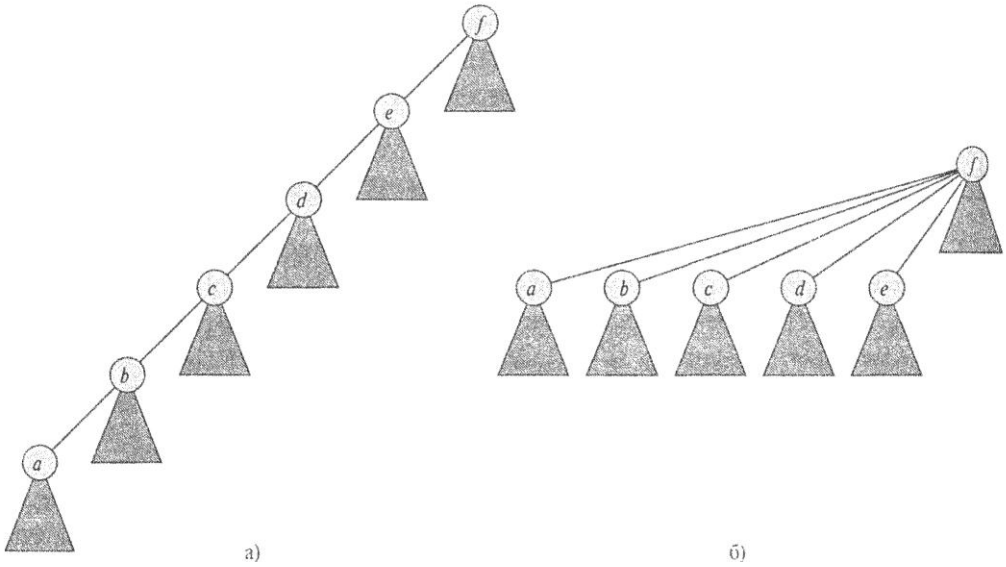


Find_Set(x) – O(1)
Union(x, y) – O(n)

Весовая эвристика – короткий список присоединяем к длинному



Первая эвристика - объединение по рангу (высота)
Вторая эвристика - сжатие пути при выполнении Find_Set




```
MAKE_SET(x)
  p[x] ← x
  rank[x] ← 0

UNION(x, y)
  LINK(FIND_SET(x), FIND_SET(y))

LINK(x, y)
  if rank[x] > rank[y]
     then p[y] ← x
     else  p[x] ← y
           if rank[x] = rank[y]
              then rank[y] ← rank[y] + 1

FIND_SET(x)
  if x ≠ p[x]
     then p[x] ← FIND_SET(p[x])
  return p[x]
```

8 Хеш-таблицы

Построение хеш-функции методом умножения

1. **H(k) = int (m (kA mod 1)) m – размер хеш-таблицы.**
2. **A – 0<A<1**

$A \approx \left(\sqrt{5}-1\right)/2$

3. **A = s/2^w w=32 (16)**
4. **kA= ks /2^w= (a2^w+b) /2^w**
5. **kA mod 1 = b /2^w**
6. **Пусть m = 2^p =>m (kA mod 1) = b /2^{w-p}**
7. **H(k) = b>>(w-p)**
8. **Пусть w=16, p=8**

9. s=int(A2^{w});
10. b=f(k*s);
11. H(k) = b>>(w-p)

Универсальное хеширование
$h_{a,b}$(k)=((ak+b) mod p) mod m; 1<a<p; 0<b<p; p>max(k)

Двойное хеширование
h (k, i) = (h_1 (k) + i*h_2 {k)) mod m,

CRC алгоритм

```
  10011011
+11001010
--------
  01010001

 10011011
-11001010
--------
 01010001

10000  : 10101
10101     1
  0101

110000  : 10101
10101     11
  11010
  10101
    1111

100000  : 10101
10101     10
  01010
  00000
    1010

1100001010
10011
11010110110000 = выровненное сообщение (1101011011 + 0000)
10011xxxxxxxxx
--------
  10011xxxxxxxx
  10011
  --------
    00001xxxxxxx
      -----
          10110xxx
          10011
          --------...
```

```
01010 xx
--------...
 10100x
 10011.
  -----.
  01110
-----
    1110 = Остаток
```

11000010 – целая часть

Загрузим регистр нулевыми битами
Дополним хвостовую часть сообщения W нулевыми битами
While (пока еще есть необработанные биты)
  Begin
    Сдвинем регистр на 1 бит влево и поместим очередной
    еще не обработанный бит из сообщения в 0 позицию регистра.
    If (из регистра был выдвинут бит со значением "1")
        Регистр = Регистр XOR Полином
  End
Теперь в регистре содержится остаток

3	**2**	**1**	**0**

t7 t6 t5 t4 t3 t2 t1 t0

t6 t5 t4 t3 t2 t1 t0 ?
XOR
t7 * (g7 g6 g5 g4 g3 g2 g1 g0)

……..

While (сообщение полностью не обработано)
Begin
    Проверим старший байт регистра
    Рассчитаем на его основе контрольный байт
    Основываясь на значении контрольного байта, выполним операцию XOR нашего полинома с различными смещениями

Сдвинем регистр на один байт влево, добавив справа новый байт из сообщения
Выполним операцию XOR содержимого регистра с суммированным полиномом
End

While (сообщение полностью не обработано)
Begin
Top = Старший_байт(Register);
Register = (Register << 8) | Новый_байт_сообщения;
Register = Register XOR Рассчитанная_таблица[Top];
End

```
r=0;
while (len--)
{
        byte t = (r >> 24) & 0xFF;
        r = (r << 8) | *p++;
        r^=table[t];
}
```

9 Поиск подстроки

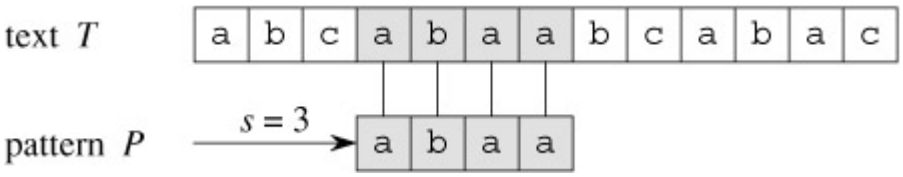


Рис 1

The naive string-matching algorithm

```
NAIVE-STRING-MATCHER(T, P)
n ← length[T]
m ← length[P]
for s ← 0 to n - m
    do if P[1 ·· m] = T[s + 1 ·· s + m]
          then print "Pattern occurs with shift" s
```

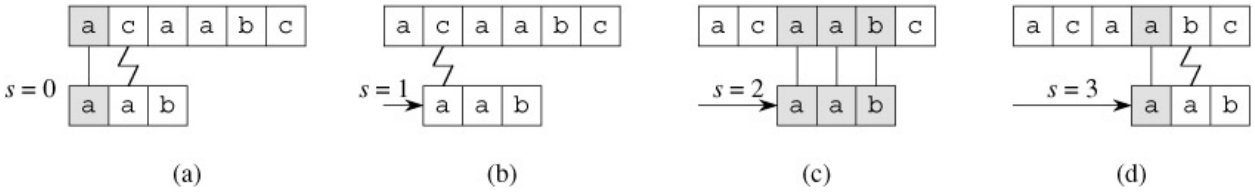


Рис 2

Procedure NAIVE-STRING-MATCHER takes time $O((n - m + 1)m)$

9.1 The Rabin-Karp algorithm

Пусть $\Sigma = \{0, 1, 2, \ldots, 9\}$

$p = P[m] + 10\,(P[m - 1] + 10(P[m - 2] + \cdots + 10(P[2] + 10P[1])\,))$.

t_0 <= $T\,[1 \cdot\cdot\, m]$ in time $\Theta(m)$

$$t_{s+1} = 10(t_s - 10^{m-1}T\{s+1\}) + T\{s+m+1\}$$

m=5
$t_s = 31415$
$t_{s+1} = 14152$

$10(31415 - 10000 \cdot 3) + 2 = 14152$.

=> сложность O(n)

$$t_{s+1} = (d(t_s - hT\{s+1\}) + T\{s+m+1\}) \bmod q$$

h=d^(m-1)

q – простое число
dq<maxint

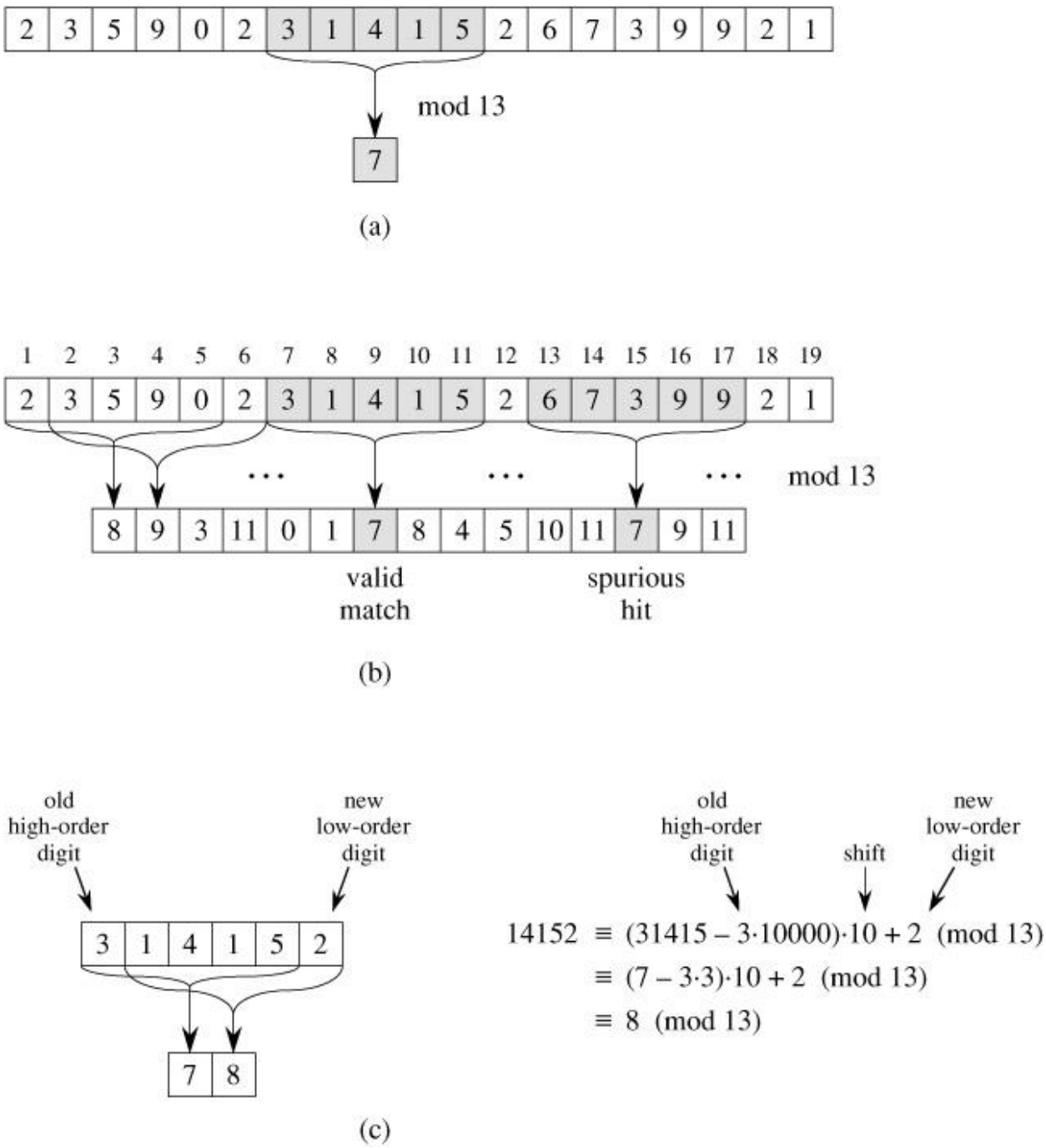


Рис 3

```
RABIN-KARP-MATCHER(T, P, d, q)
n ← length[T]
m ← length[P]
h ← d^(m-1) mod q
p ← 0
t0 ← 0
for i ← 1 to m                ▷ Preprocessing.
    do p ← (dp + P[i]) mod q
       t0 ← (dt0 + T[i]) mod q
for s ← 0 to n - m            ▷ Matching.
    do if p = ts
          then if P[1 ·· m] = T [s + 1 ·· s + m]
                  then print "Pattern occurs with shift" s
       if s < n - m
```

```
14            then t_{s+1} ← (d(t_s - T[s + 1]h) + T[s + m + 1]) mod q
```

=> сложность O(n) в среднем и O(n*m) в худшем случае.

9.2 String matching with finite automata

Finite automata

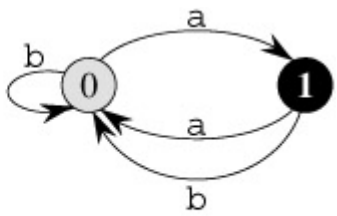
A ***finite automaton*** M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

- Q is a finite set of ***states***,
- $q_0 \in Q$ is the ***start state***,
- $A \subseteq Q$ is a distinguished set of ***accepting states***,
- Σ is a finite ***input alphabet***,
- δ is a function from $Q \times \Sigma$ into Q, called the ***transition function*** of M.

δ(q,a)

state	input a	input b
0	1	0
1	0	0

(a)



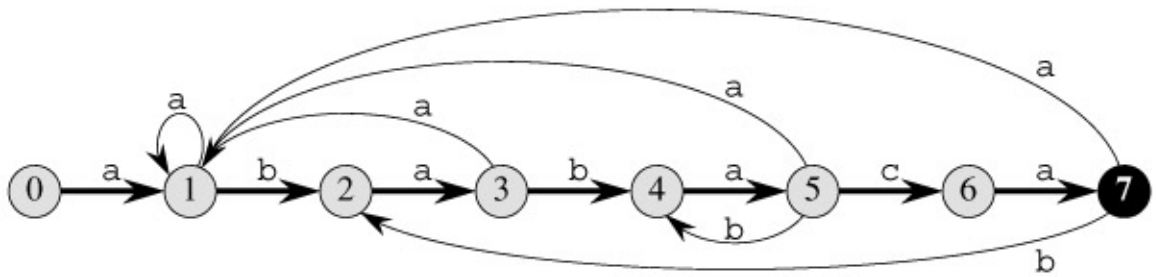
(b)

Рис 4

final-state function $\varphi(w)$ – состояние, в которое перейдет автомат прочитав строку w.

$$\varphi(\varepsilon) = q_0,$$
$$\varphi(wa) = \delta(\varphi(w), a)$$

String-matching automata



(a)

state	input a	input b	input c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

(b)

i	—	1	2	3	4	5	6	7	8	9	10	11
$T[i]$	—	a	b	a	b	a	b	a	c	a	b	a
state $\phi(T_i)$	0	1	2	3	4	5	4	5	6	7	2	3

(c)

Рис 5

$\sigma(x) = \max\ \{k : P_k \sqsupset x\}.$

$\delta(q,a) = \sigma(P_q a)$

```
FINITE-AUTOMATON-MATCHER(T, δ, m)
n ← length[T]
q ← 0
for i ← 1 to n
    do q ← δ(q, T[i])
       if q = m
          then print "Pattern occurs with shift" i - m
```

O(n)

```
COMPUTE-TRANSITION-FUNCTION(P, Σ)
m ← length[P]
for q ← 0 to m
    do for each character a ∈ Σ
          do k ← min(m + 1, q + 2)
             repeat k ← k - 1
               until P_k ⊐ P_q a
             δ(q, a) ← k
return δ
```

$O(m^3 |\Sigma|)$

9.3 The Knuth-Morris-Pratt algorithm

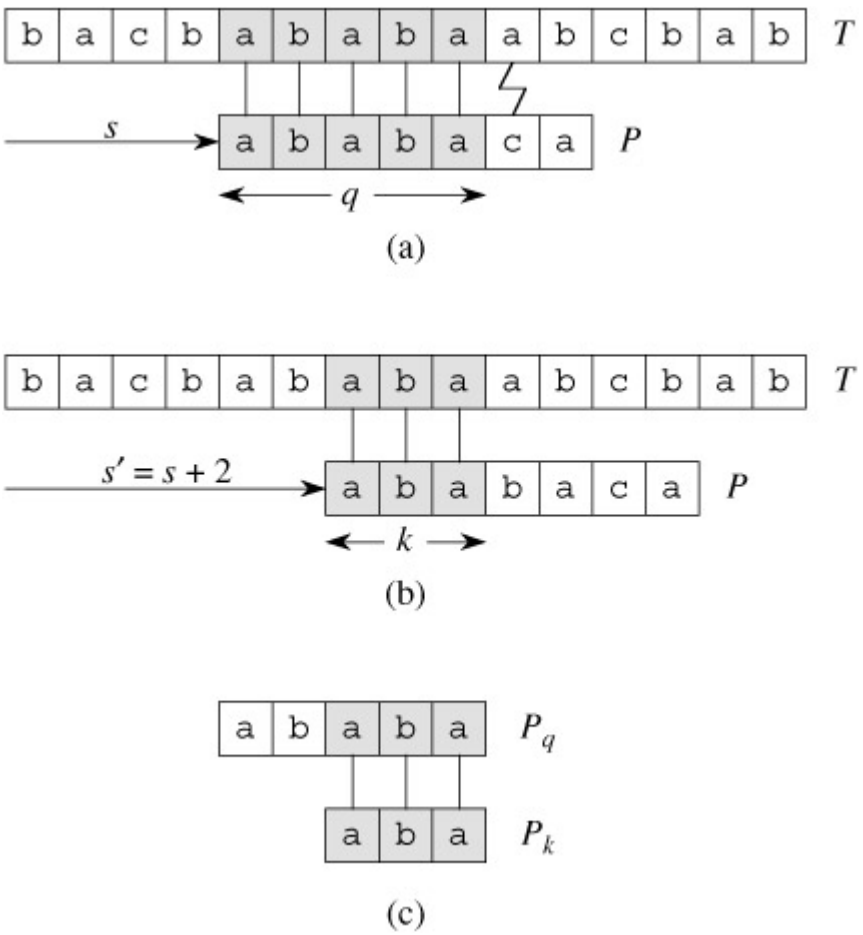


Рис 6

Префикс-функция

k=π[q] = max {k : $k < q$ and $P_k \sqsupset P_q$}.

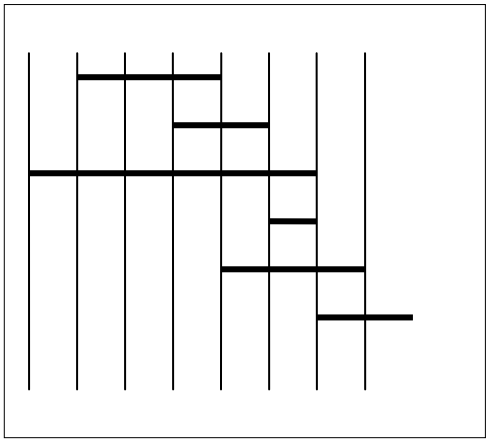
```
KMP-MATCHER(T, P)
 1 n ← length[T]
 2 m ← length[P]
 3 π ← COMPUTE-PREFIX-FUNCTION(P)
 4 q ← 0                          ▹Number of characters matched.
 5 for i ← 1 to n                 ▹Scan the text from left to right.
 6      do while q > 0 and P[q + 1] ≠ T[i]
 7             do q ← π[q]     ▹Next character does not match.
 8         if P[q + 1] = T[i]
 9            then q ← q + 1      ▹Next character matches.
10         if q = m                     ▹Is all of P matched?
11            then print "Pattern occurs with shift" i - m
12                 q ← π[q]     ▹Look for the next match.
```

O(n+m)

```
COMPUTE-PREFIX-FUNCTION(P)
m ← length[P]
π[1] ← 0
k ← 0
for q ← 2 to m
     do while k > 0 and P[k + 1] ≠ P[q]
            do k ← π[k]
        if P[k + 1] = P[q]
           then k ← k + 1
        π[q] ← k
return π
```

10 Жадные алгоритмы

Задача о выборе процессов



$f_0 \leqslant f_1 \leqslant \cdots \leqslant f_n \leqslant f_{n+1}$.

$S_{ij} = \{a_k \in S : f_i \leqslant s_k < f_k \leqslant s_j\}$

c[i,j] — количество процессов в подмножестве максмаксимального размера, состоящем из взаимно совместимых процессов в задаче S_{ij}.

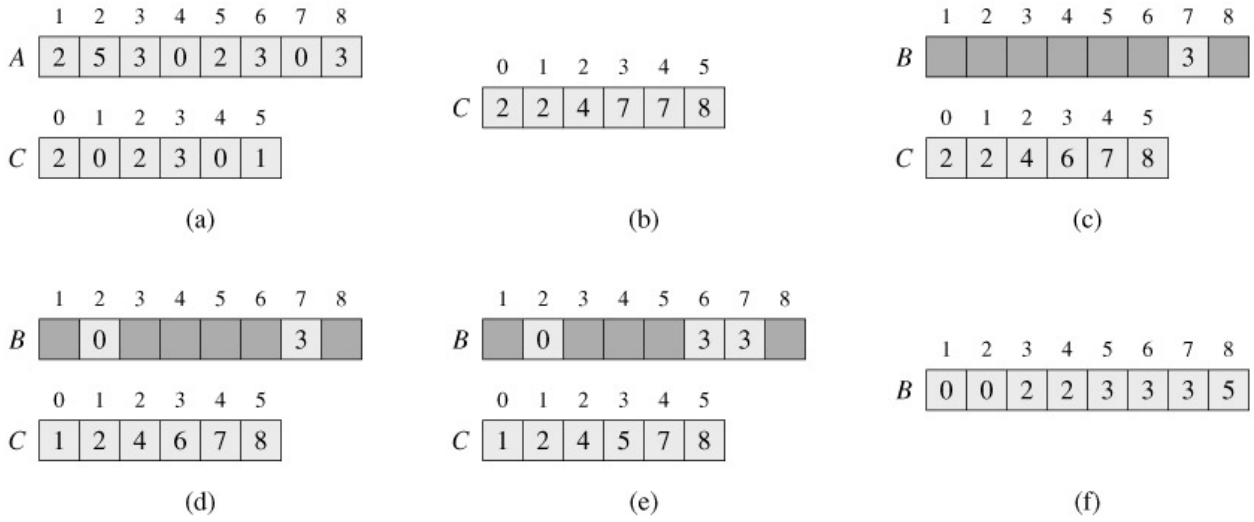
$$c[i,j] = \begin{cases} 0 & \text{при } S_{ij} = \emptyset, \\ \max\limits_{\substack{i<k<j \text{ и} \\ a_k \in S_{ij}}} \{c[i,k] + c[k,j] + 1\} & \text{при } S_{ij} \neq \emptyset. \end{cases}$$

11 Сортировки за линейное время

Сортировка подсчетом

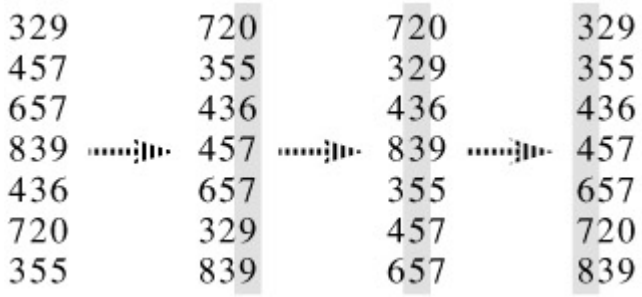
```
COUNTING-SORT(A, B, k)
 for i ← 0 to k
    do C[i] ← 0
 for j ← 1 to length[A]
```

```
    do C[A[j]] ← C[A[j]] + 1
▷ C[i] now contains the number of elements equal to i.
for i ← 1 to k
    do C[i] ← C[i] + C[i - 1]
▷ C[i] now contains the number of elements less than or equal to i.
for j ← length[A] downto 1
    do B[C[A[j]]] ← A[j]
       C[A[j]] ← C[A[j]] - 1
```



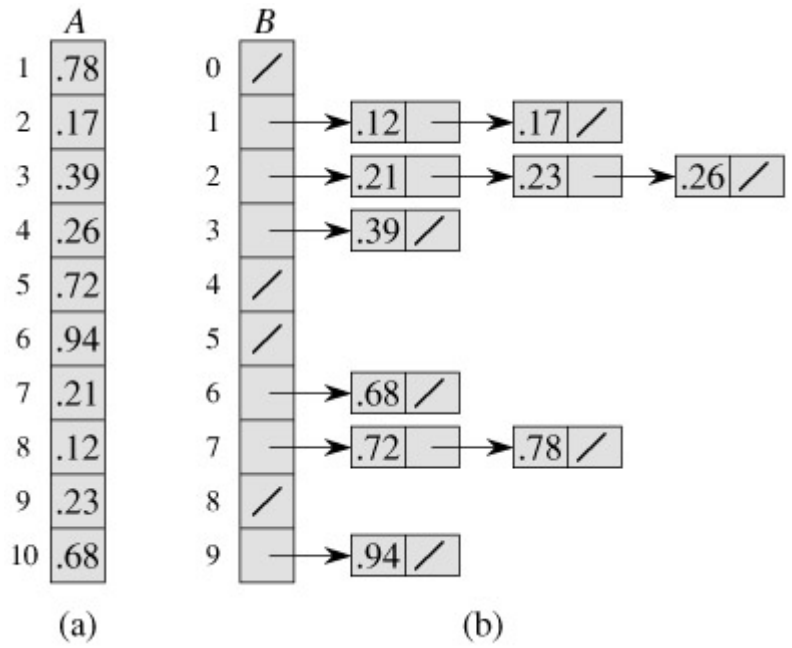
Цифровая сортировка

```
RADIX-SORT(A, d)
for i ← 1 to d
    do use a stable sort to sort array A on digit i
```



Сортировка вычерпыванием

```
BUCKET-SORT(A)
n ← length[A]
for i ← 1 to n
    do insert A[i] into list B[⌊n A[i]⌋]
for i ← 0 to n - 1
    do sort list B[i] with insertion sort
concatenate the lists B[0], B[1], . . ., B[n - 1] together in order
```



12 Быстрое преобразование Фурье (FFT)

Дескретное преобразование Фурье:

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

$$\begin{aligned} y_k &= A(\omega_n^k) \\ &= \sum_{j=0}^{n-1} a_j \omega_n^{kj} . \end{aligned}$$

$\omega_n^k = \exp(2\pi ki/n)$, **k = 0…n-1**

$\boldsymbol{e^{iu} = \cos(u) + i\sin(u)}$.

$\boldsymbol{y = (y_0, y_1,..., y_{n-1})}$, $\boldsymbol{a = (a_0, a_1,..., a_{n-1})}$

$$A(x) = A^{[0]}(x^2) + xA^{[1]}(x^2) , \qquad (1)$$

$$A^{[0]}(x) = a_0 + a_2x + a_4x^2 + \cdots + a_{n-2}x^{n/2-1}, \qquad (2)$$

$$A^{[1]}(x) = a_1 + a_3x + a_5x^2 + \cdots + a_{n-1}x^{n/2-1}. \qquad (3)$$

$$(\omega_n^0)^2, (\omega_n^1)^2, \ldots, (\omega_n^{n-1})^2 , \qquad \textbf{(4)}$$

$$\omega_n^{2k} = \exp(2*2\pi ki/n) = \exp(2\pi ki/(n/2)) = \omega_{n/2}^k, \qquad \textbf{(5)}$$

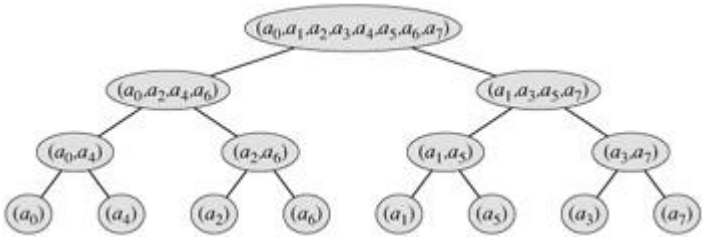
$$\omega_{n/2}^{k+n/2} = \exp(2\pi ki/(n/2)) * \exp(2\pi i) = \omega_{n/2}^k, \qquad \textbf{(6)}$$

$$\omega_n^{k+n/2} = \exp(2\pi ki/n) * \exp(\pi i) = -\omega_n^k, \qquad \textbf{(7)}$$

RECURSIVE-FFT(*a*)

$n \leftarrow length[a]$ ▷ n is a power of 2.
if $n = 1$
then return a
$w_n \leftarrow e^{\pi i/n}$
$w \leftarrow 1$
$a^{[0]} \leftarrow (a_0, a_2, ..., a_{n-2})$
$a^{[1]} \leftarrow (a_1, a_3, ..., a_{n-1})$
$y^{[0]} \leftarrow$ RECURSIVE-FFT($a^{[0]}$)
$y^{[1]} \leftarrow$ RECURSIVE-FFT($a^{[1]}$)
for $k \leftarrow 0$ to $n/2 - 1$
do
$y_k \leftarrow y_k^{[0]} + \omega\, y_k^{[1]}$
$y_{k+(n/2)} \leftarrow y_k^{[0]} - \omega\, y_k^{[1]}$
$w \leftarrow w\, w_n$
return y

Без рекурсии:




```
ITERATIVE-FFT (a)
BIT-REVERSE-COPY (a, A)
n ← length[a]        ▷ n is a power of 2.
for s ← 1 to lg n
     do m ← 2^s
        ω_m ← e^(2πi/m)
        for k ← 0 to n - 1 by m
             do ω ← 1
                for j ← 0 to m/2 - 1
                    do t ← ωA[k + j + m/2]
                       u ← A[k + j]
                       A[k + j] ← u + t
                       A[k + j + m/2] ← u - t
                       ω ← ω ω_m
```

13 Вопросы к экзамену 2010

1. Обратная польская запись. Построение и вычисление.
2. Управление динамической памятью.
3. Растеризация невыпуклого многоугольника

4. Топологическая сортировка.
5. Задача о “Ханойской башне”. Вывод формулы сложности.
6. Синтаксический анализ арифметического выражения
7. Перебор
8. Перестановки
9. Сокращение перебора. N ферзей.
10. Сокращение перебора. Задача о ходах коня.
11. Сокращение перебора. Задача о коммивояжере.
12. Сокращение перебора. Раскраска карты.
13. Сокращение перебора. Распространение ограничений и изменение порядка перебора
14. Сокращение перебора. Укладка рюкзака.
15. Алгоритмы обхода бинарных деревьев, включая обход в ширину.
16. Сильноветвящиеся деревья и графы
17. Деревья двоичного поиска
18. Деревья с дополнительной информацией (Добавление с поддержкой числа узлов)
19. Деревья с дополнительной информацией (К-й наименьший и порядковый номер)
20. Динамическое программирование. Конвейер.
21. Динамическое программирование. Наименьшая сумма чисел на пути
22. Динамическое программирование. Умножение матриц
23. Динамическое программирование. Самая длинная общая подпоследовательность
24. АВЛ дерево
25. Красно-черное дерево
26. Treaps
27. Скошенные деревья
28. 2-3 дерево
29. 2-3+ дерево
30. B – деревья
31. Кучи (поддержка свойств и построение)
32. Очередь с приоритетами
33. Huffman коды
34. Кучи на 2-3+ деревьях
35. Биномиальные кучи (определение)
36. Биномиальные кучи (объединение, забрать min ключ, удаление)
37. Фибоначевы кучи
38. Структуры данных для непересекающихся множеств
39. Хеш-таблицы
40. CRC алгоритм
41. Табличный CRC алгоритм
42. Поиск подстроки. Алгоритм Рабина-Карпа.
43. Поиск подстроки с конечным автоматом.
44. Поиск подстроки. Алгритм Кнута-Мориса-Прата.
45. Быстрое преобразование Фурье