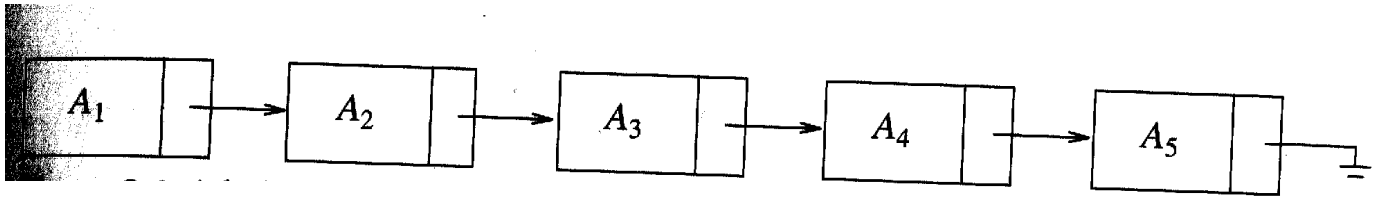
СПИСЪЦИ

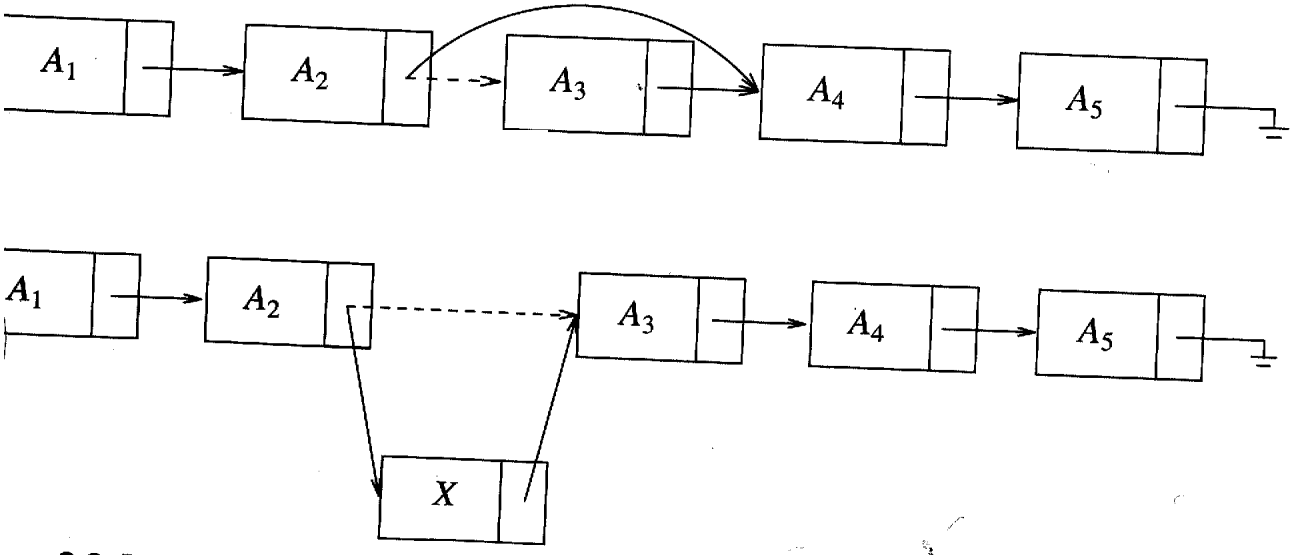


понятия:

- ***дефиниция;***
- ***типове: едносвързни, двойносвързани, кръгово-свързани, множествени списъкови структури***
- ***размер;***
- ***празен списък;***
- ***позиция;***
- ***операции над списък: makeEmpty, insert, remove, find, print, sort, next, previous, first, last***
- ***реализации: чрез масив, чрез структура и указатели. Анализ на предимства и недостатъци.***

Свързани списъци

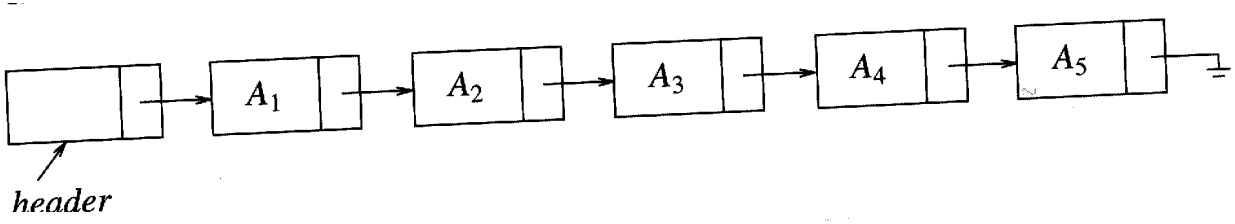
- изтриване на елемент:



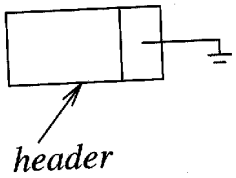
- вмъкване на елемент

особености по реализацията на списъци

-затруднения при вмъкване в началото, изтриване на ел. от началото и изобщо изтриване на ел.
- подобрение с добавяне на header:



свързан списък с header елемент



празен списък с header елемент

Шаблони в C++ (Templates) : a bit of theory

-Function templates
-Class templates

template specialization
(generic programming in .NET)

Based on templates is constructed the Standard Template Library (STL) containing : container classes, integrators and algorithms

1. Function templates

The compiler generates separate source-code functions (named specializations) to handle each function call appropriately.
(The same can be performed using macros (#define…). However, macros are not able to perform type checking)

```
template<typename T >
Or
template< class ElementType >
Or
template< typename Border, typename fillType >
```

Means 'any fundamental type or user-defined type'

Example: let construct a **function template**: printArray()

```
// function template printArray definition
template< typename T >
void printArray( const T * const array, int count )
{   for ( int i = 0; i < count; i++ )      cout << array[ i ] << " ";   cout << endl;} // end function template int
main()
{  const int aCount = 5; // size of array a
   const int bCount = 7; // size of array b
   const int cCount = 6; // size of array c

   int a[ aCount ] =              { 1, 2, 3, 4, 5 };
   double b[ bCount ] =           { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7 };
   char c[ cCount ] =             "HELLO";                              // 6th position for null

   cout << "Array a contains:" << endl;

   // call integer function-template specialization
   printArray( a, aCount );
   cout << "Array b contains:" << endl;

   // call double function-template specialization
   printArray( b, bCount );
   cout << "Array c contains:" << endl;

   // call character function-template specialization
   printArray( c, cCount );
} // end main
```

So:

```
void printarray(const int * const, int);
void printarray( const double * const, int);
```

2. Class templates

```
template< typename T >
class Stack
{public:
   Stack( int = 10 ); // default constructor (Stack size 10)
   ~Stack()
   { delete [] stackPtr;          // deallocate internal space for Stack
   }                              // end ~Stack destructor

   bool push( const T & );      // push an element onto the Stack
   bool pop( T & ); // pop an element off the Stack

   // determine whether Stack is empty
   bool isEmpty() const
   {      return top == -1;    } // end function isEmpty

   // determine whether Stack is full
   bool isFull() const
   {      return top == size - 1;    } // end function isFull

private:
   int size;                      // # of elements in the stack
   int top;                       // location of the top element (-1 means empty)
   T *stackPtr;                   // pointer to internal representation of the Stack
};                                // end class template Stack
```

```
// constructor template
template< typename T >
Stack< T >::Stack( int s )
   : size( s > 0 ? s : 10 ), // validate size
     top( -1 ), // Stack initially empty
     stackPtr( new T[ size ] ) // allocate memory for elements
{}   // empty body for  Stack constructor template

// push element onto Stack;
// if successful, return true; otherwise, return false
template< typename T >
bool Stack< T >::push( const T &pushValue )
{
   if ( !isFull() )
   {
      stackPtr[ ++top ] = pushValue; // place item on Stack
      return true; // push successful
   } // end if

   return false; // push unsuccessful
} // end function template push

// pop element off Stack;
// if successful, return true; otherwise, return false
template< typename T >
bool Stack< T >::pop( T &popValue )
{…}
……..
```

Using class template Stack<T>

```
// Stack class template test program.
#include <iostream>
#include "Stack.h"                          // Stack class template definition
using namespace std;

int main()
{
  Stack< double > doubleStack( 5 );         // size 5
  double doubleValue = 1.1;

  cout << "Pushing elements onto doubleStack\n";

  // push 5 doubles onto doubleStack
  while ( doubleStack.push( doubleValue ) )
  {
    cout << doubleValue << ' ';
    doubleValue += 1.1;
  }                                         // end while

  cout << "\nStack is full. Cannot push " << doubleValue
    << "\n\nPopping elements from doubleStack\n";

  // pop elements from doubleStack
  while ( doubleStack.pop( doubleValue ) )
    cout << doubleValue << ' ';

……….
```

Templates & inheritance

- A class template can be derived from a class-template specialization
- A class template can be derived from a class-nontemplate
- A class template-specialization can be derived from a class-template specialization
- A non-template class can be derived from a class-template specialization

Now to return to complex data structures:

-декларация на абстрактен списък, съставен от 3 класа:

- самия списък;
- възел;
- позиция (iterator-class).

```
template <class Object>
class List;          // Incomplete declaration.

template <class Object>
class ListItr;       // Incomplete declaration.

template <class Object>
class ListNode
{
    ListNode( const Object & theElement = Object( ), ListNode * n = NULL )
      : element( theElement ), next( n ) { }

    Object    element;
    ListNode *next;

    friend class List<Object>;
    friend class ListItr<Object>;
};
```

-реализация на класа – iterator: съдържа методи за прохожане;
- конструкторът му е private с friend class – List.

```
template <class Object>
class ListItr
{
  public:
    ListItr( ) : current( NULL )        // за общо използване
    {
    }

    bool isPastEnd( ) const
    {
        return current == NULL;
    }

    void advance( )
    {
        if( !isPastEnd( ) )
            current = current->next;
    }

    const Object & retrieve( ) const
    {
        if( isPastEnd( ) )
            throw BadIterator( );
        return current->element;
    }

  private:
    ListNode<Object> *current;     // Current position

    ListItr( ListNode<Object> *theNode ) : current( theNode ) { }

    friend class List<Object>;     // Grant access to constructor
};
```

за използване от List

- декларация на List клас:

```
template <class Object>
class List
{
  public:
    List( );
    List( const List & rhs );
    ~List( );

    bool isEmpty( ) const;
    void makeEmpty( );
    ListItr<Object> zeroth( ) const;
    ListItr<Object> first( ) const;
    void insert( const Object & x, const ListItr<Object> & p);
    ListItr<Object> find( const Object & x ) const;
    ListItr<Object> findPrevious( const Object & x ) const
    void remove( const Object & x );

    const List & operator=( const List & rhs );

  private:
    ListNode<Object> *header;      // инициира се в конструктора после
};
```

- дефиниция на методи на List класа:


```
template <class Object>
List<Object>::List( )
{
    header = new ListNode<Object>;
}

/**
 * Test if the list is logically empty.
 * Return true if empty, false otherwise.
 */
template <class Object>
bool List<Object>::isEmpty( ) const
{
    return header->next == NULL;
}

/**
 * Return an iterator representing the header node.
 */
template <class Object>
ListItr<Object> List<Object>::zeroth( ) const
{
    return ListItr<Object>( header );
}
```

(*continued*)

```
/**
 * Return an iterator representing the first node in the list.
 * This operation is valid for empty lists.
 */
template <class Object>
ListItr<Object> List<Object>::first( ) const
{
    return ListItr<Object>( header->next );
}
```

! - **функции от класа: за връщане на първи елемент и за print**

```
// Simple print function
template <class Object>
void printList( const List<Object> & theList )
{
    if( theList.isEmpty( ) )
        cout << "Empty list" << endl;
    else
    {
        ListItr<Object> itr = theList.first( );
        for( ; !itr.isPastEnd( ); itr.advance( ) )
            cout << itr.retrieve( ) << " ";
    }

    cout << endl;
}
```

-необходими ли са 3 класа в реализацията на списъка?

класът ‘iterator’ има няколко функции:

1. абстрактните понятия ‘списък’ и ‘позиция’ са различими;
2. списъкът може да се обработва едновременно от няколко места, чрез няколко iterator класа.

-методът find() може да се реализира рекурсивно, но има по-добър подход (виж по-долу);

- почти всички методи се оценяват с O(1), find() и findprevious() - с O(N);

```
         * Return iterator corresponding to the first node containing an item x.
         * Iterator isPastEnd if item is not found.
         */
        template <class Object>
        ListItr<Object> List<Object>::find( const Object & x ) const
        {
/* 1*/      ListNode<Object> *itr = header->next;

/* 2*/      while( itr != NULL && itr->element != x )
/* 3*/          itr = itr->next;

/* 4*/      return ListItr<Object>( itr );
        }
```

```
 * Remove the first occurrence of an item x.
 */
template <class Object>
void List<Object>::remove( const Object & x )
{
    ListItr<Object> p = findPrevious( x );

    if( p.current->next != NULL )
    {
        ListNode<Object> *oldNode = p.current->next;
        p.current->next = p.current->next->next;  // Bypass deleted node
        delete oldNode;
    }
}
```

```
    /**
     * Return iterator prior to the first node containing an item x.
     */
    template <class Object>
    ListItr<Object> List<Object>::findPrevious( const Object & x ) const
    {
1*/     ListNode<Object> *itr = header;

2*/     while( itr->next != NULL && itr->next->element != x )
3*/         itr = itr->next;

4*/     return ListItr<Object>( itr );
    }
```

```
/**
 * Insert item x after p.
 */
template <class Object>
void List<Object>::insert( const Object & x, const ListItr<Object> & p);
{
    if( p.current != NULL )
        p.current->next = new ListNode<Object>( x, p.current->next );
}
```

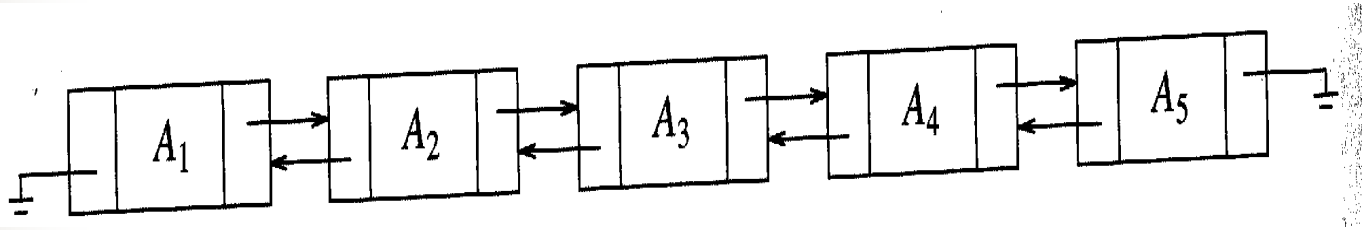
работа с паметта:

- особености при работа с указатели: временно съхраняваме указат. към ненужен обект и го пазим докато окончателно приключим работата с обекта (пример: remove() в който delete oldnode е в края);
- работата с памет е изнесена в специализирани методи; главно в деструкция.

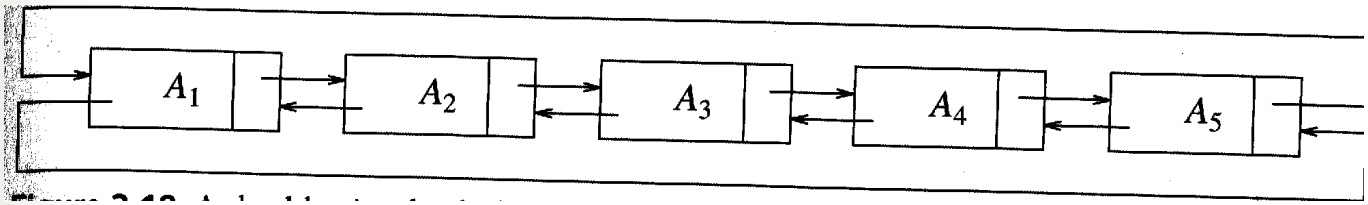
избягваме delete, колкото е възможно

```
/**
 * Make the list logically empty.
 */
template <class Object>
void List<Object>::makeEmpty( )
{
    while( !isEmpty( ) )
        remove( first( ).retrieve( ) );
}

/**
 * Destructor
 */
template <class Object>
List<Object>::~List( )
{
    makeEmpty( );
    delete header;
}
```

Двойно свързани списъци



Кръгово-свързани списъци



Приложни аспекти на списъците

1. ***списъци в представянето на полином (абстрактен тип – полином, с положителни експоненти)***

пр: $P_1(x) = 10x^{1000} + 5x^{14} + 1$

$P_2(x) = 3x^{1990} – 2x^{1492} + 11x + 5$

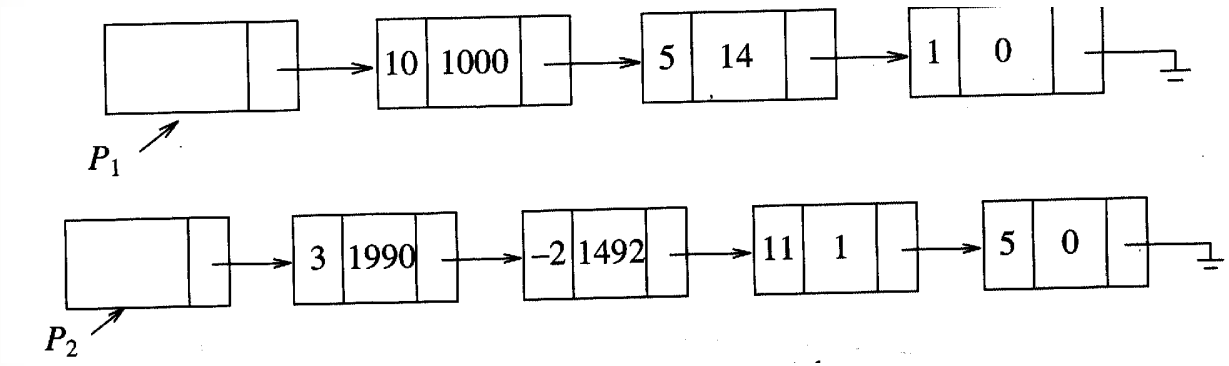
При коефициенти a_i **!= 0** в преобладаваща степен: удобна е реализация през масив. и процедури за събиране, изваждане, умножение … над полиноми. Например:

```
class Polynomial
{
    public:
        Polynomial();
        void insertTerm(int coef, int exp);
        void zeroPolynomial();
        Polynomial operator+(const Polynomial &rhs) const;
        ....
        private:
        vector<int> coeffArray;
        int highPower;
};
```

недостатъци на реализацията с масив:

- време за начална инициализация на масива с 0;
- операция умножение, напр. е с време, пропорционално на произведението на степените на двата полинома (независимо, че повечето ел. в матриците са 0.
- разхищение на памет.

алтернативен подход – използване на свързан списък:



ето декларацията:

```
class Literal
{
  private:
      // Various constructors
    int coefficient;
    int exponent;
    friend class Polynomial;
};


class Polynomial
{
  public:
    Polynomial( );
    void insertTerm( int coef, int exp );

    void zeroPolynomial( );
    Polynomial operator+( const Polynomial & rhs ) const;
    Polynomial operator*( const Polynomial & rhs ) const;
    void print( ostream & out ) const;

  private:
    List<Literal> terms;
};
```

2. сложни приложни списъкови структури: - многосвързани списъци

Задача:

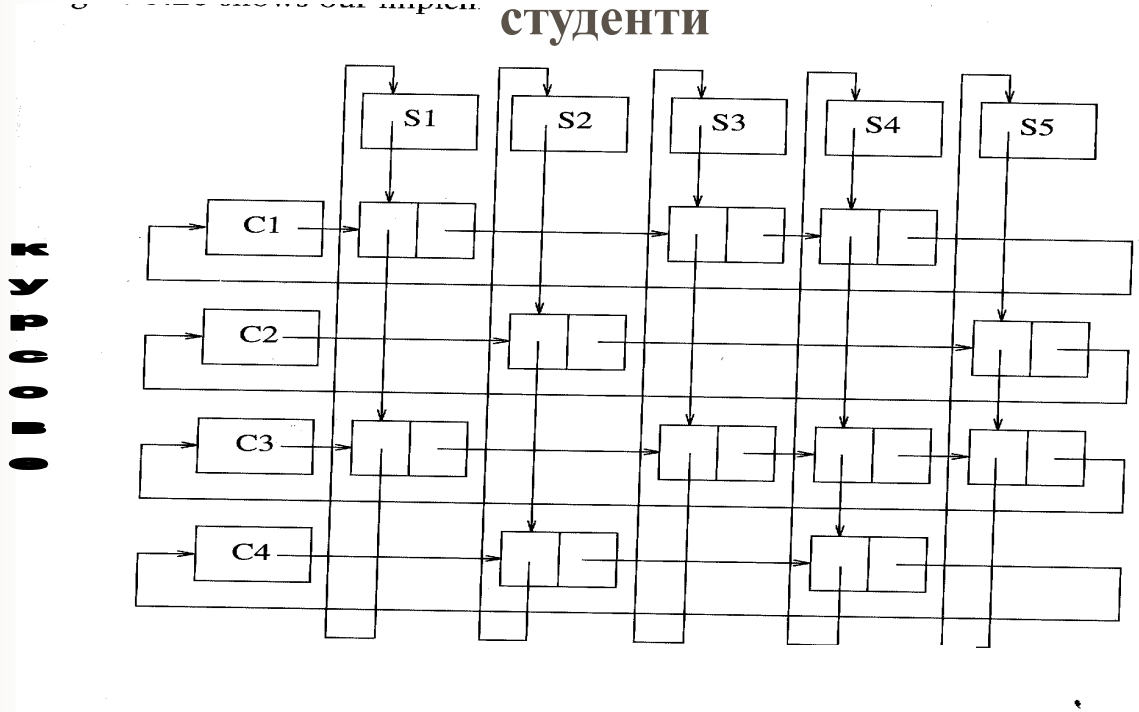
Имаме университетска структура с 40 000 студента и 2500 курса. Необходима е структура , предпоставяща лесно генериране на отчети. Напр:

- списък на студенти по предмети

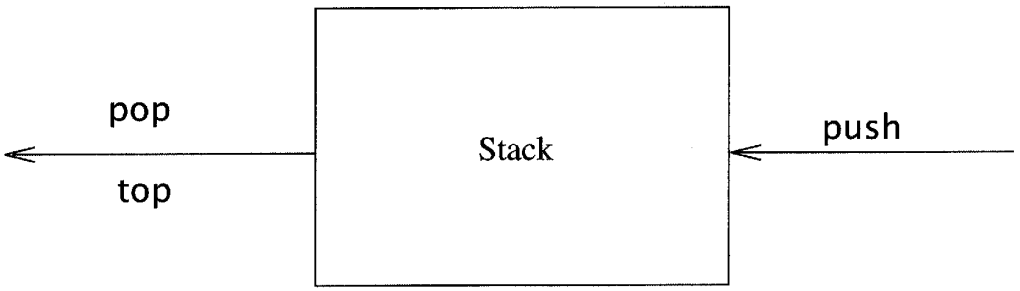
- списък на изучавани от студент курсове.

Първа реализация – през 2-мерен масив: студенти ← → курсове.
за горните стойности – масив с 100 000 000 клетки.
При средно записване по 3 курса от студент – само 120 000 клетки носят информация (0.1 %).

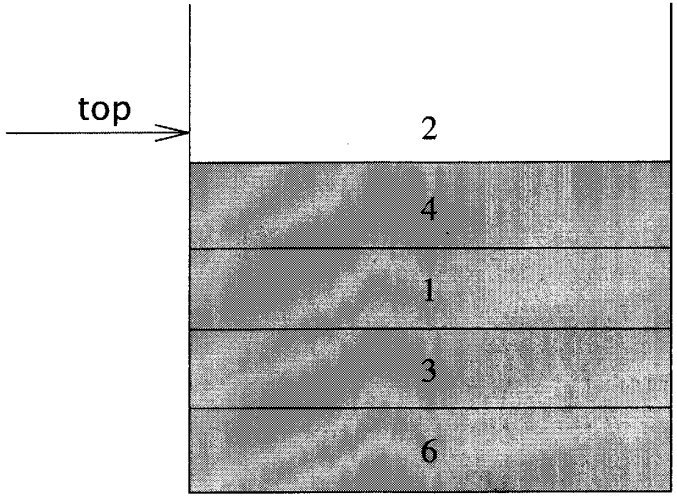
Друга реализация – матричен (двойносвързан) списък:



Абстрактен тип - стек



Stack model: input to a stack is by `push`, output is by `pop` and `top`



реализация на стек през свързан списък

```
template <class Object>
class Stack
{
  public:
    Stack( );
    Stack( const Stack & rhs );
    ~Stack( );

    bool isEmpty( ) const;
    bool isFull( ) const;
    const Object & top( ) const;

    void makeEmpty( );
    void pop( );
    void push( const Object & x );
    Object topAndPop( );

    const Stack & operator=( const Stack & rhs );

  private:
    struct ListNode
    {
        Object    element;
        ListNode *next;

        ListNode( const Object & theElement, ListNode * n = NULL )
          : element( theElement ), next( n ) { }
    };

    ListNode *topOfStack;
};
```

```
/**
 * Construct the stack.
 */
template <class Object>
Stack<Object>::Stack( )
{
    topOfStack = NULL;
}
```

```
/**
 * Destructor.
 */
template <class Object>
Stack<Object>::~Stack( )
{
    makeEmpty( );
}
```

```
/**
 * Test if the stack is logically full.
 * Return false always, in this implementation.
 */
template <class Object>
bool Stack<Object>::isFull( ) const
{
    return false;
}
```

```
/**
 * Test if the stack is logically empty.
 * Return true if empty, false otherwise.
 */
template <class Object>
bool Stack<Object>::isEmpty( ) const
{
    return topOfStack == NULL;
}
```

```
/**
 * Make the stack logically empty.
 */
template <class Object>
void Stack<Object>::makeEmpty( )
{
    while( !isEmpty( ) )
        pop( );
}
```

```
 * Insert x into the stack.
 */
template <class Object>
void Stack<Object>::push( const Object & x )
{
    topOfStack = new ListNode( x, topOfStack );
}
```

Routine to push onto a stack—linked list implementation

```
/**
 * Get the most recently inserted item in the stack.
 * Return the most recently inserted item in the stack
 * or throw an exception if empty.
 */
template <class Object>
const Object & Stack<Object>::top( ) const
{
    if( isEmpty( ) )
        throw Underflow( );
    return topOfStack->element;
}
```

```
/**
 * Remove the most recently inserted item from the stack.
 * Throw the Underflow exception if the stack is empty.
 */
template <class Object>
void Stack<Object>::pop( )
{
    if( isEmpty( ) )
        throw new Underflow( );

    ListNode *oldTop = topOfStack;
    topOfStack = topOfStack->next;
    delete oldTop;
}
```

Routine to pop from a stack—linked list implementation

```
/**
 * Return and remove the most recently inserted item from the sta
 * Throw the Underflow exception if the stack is empty.
 */
template <class Object>
Object Stack<Object>::topAndPop( )
{
    Object topItem = top( );
    pop( );
    return topItem;
```

реализация на стек през масив:

```
template <class Object>
class Stack
{
  public:
    explicit Stack( int capacity = 10 );

    bool isEmpty( ) const;
    bool isFull( ) const;
    const Object & top( ) const;

    void makeEmpty( );
    void pop( );
    void push( const Object & x );
    Object topAndPop( );

  private:
    vector<Object> theArray;
    int            topOfStack;
};
```

връща не указател, а елемента, който последно е вкаран

приложения със стек :

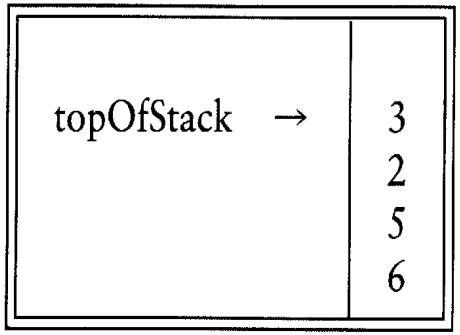
управлява приоритетите;
удобен за машинна реализация;

1. постфиксен запис :

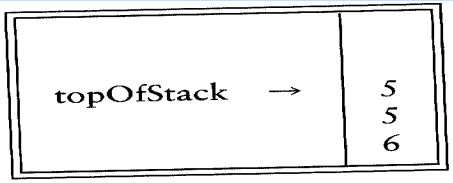
For instance, the postfix expression

6 5 2 3 + 8 * + 3 + *

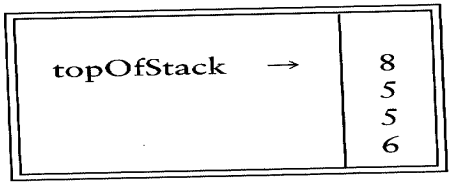
is evaluated as follows: The first four symbols are placed on the stack. The resulting stack is



Next a '+' is read, so 3 and 2 are popped from the stack and their sum, 5, is pushed.

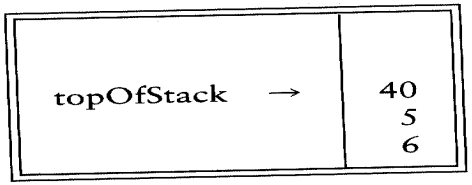


Next 8 is pushed.

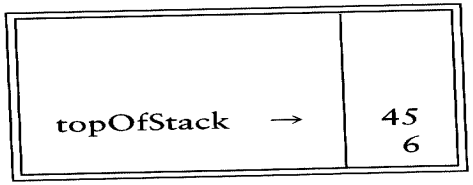


6523+8*+3+*

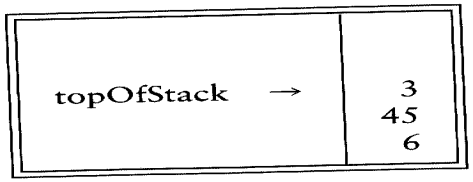
Now a '*' is seen, so 8 and 5 are popped and $5 * 8 = 40$ is pushed.



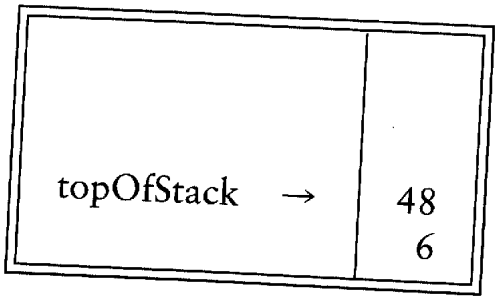
Next a '+' is seen, so 40 and 5 are popped and $5 + 40 = 45$ is pushe



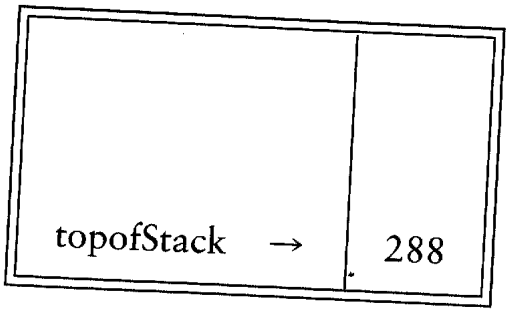
Now, 3 is pushed.



Next '+' pops 3 and 45 and pushes $45 + 3 = 48$.



Finally, a '*' is seen and 48 and 6 are popped; the result, $6 * 48 = 288$, is pu



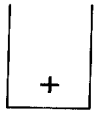
2. преобразуване (през стек) от префиксен към постфикен запис:

нека имаме:

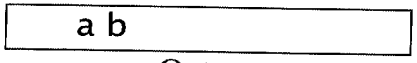
a + b * c + (d * e + f) * g

правила:
1.при ‘)’ извличаме от стека до ‘(‘
2. при прочитане на ‘операция’ извличаме от стека до достигане на по-ниско приоритетна операция.
3. достигайки края на входния поток, изпразваме стека, пишейки всичко в изходния поток.

To see how this algorithm performs, we will convert the long infix expressic above into its postfix form. First, the symbol a is read, so it is passed through to tl output. Then + is read and pushed onto the stack. Next b is read and passed throug to the output. The state of affairs at this juncture is as follows:

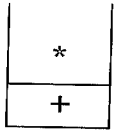


Stack

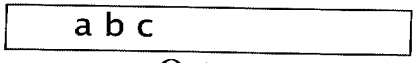


Output

Next a * is read. The top entry on the operator stack has lower precedence than so nothing is output and * is put on the stack. Next, c is read and output. Thus fa we have

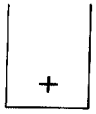


Stack

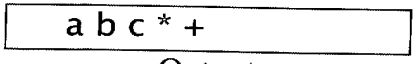


Output

The next symbol is a +. Checking the stack, we find that we will pop a * and place it on the output; pop the other +, which is not of *lower* but equal priority, on the stack; and then push the +.



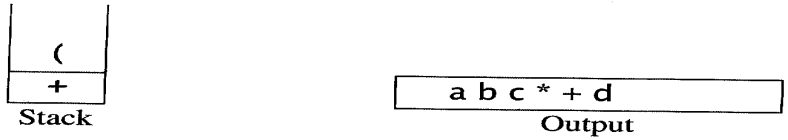
Stack



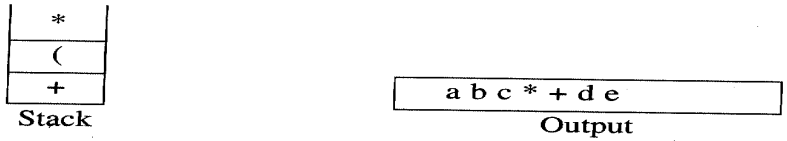
Output

a+b*c+(d*e+f)*g

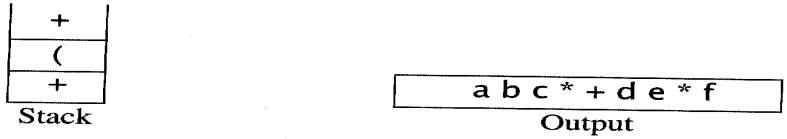
The next symbol read is a (, which, being of highest precedence, is placed on the stack. Then d is read and output.



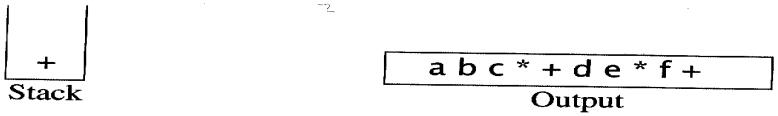
We continue by reading a *. Since open parentheses do not get removed except when a closed parenthesis is being processed, there is no output. Next, e is read and output.



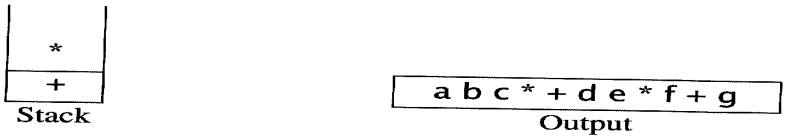
The next symbol read is a +. We pop and output * and then push +. Then we read and output f.



Now we read a), so the stack is emptied back to the (. We output a +.



We read a * next; it is pushed onto the stack. Then g is read and output.



The input is now empty, so we pop and output symbols from the stack until it is empty.

стек

abc * + de * f + g * +
това е изходът

3. викане на функции:

предаване на параметри;
специален случай при рекурсивни повиквания (фрейм)
пример с опашна рекурсия и лоша структура (напр. при 20000 елемента):

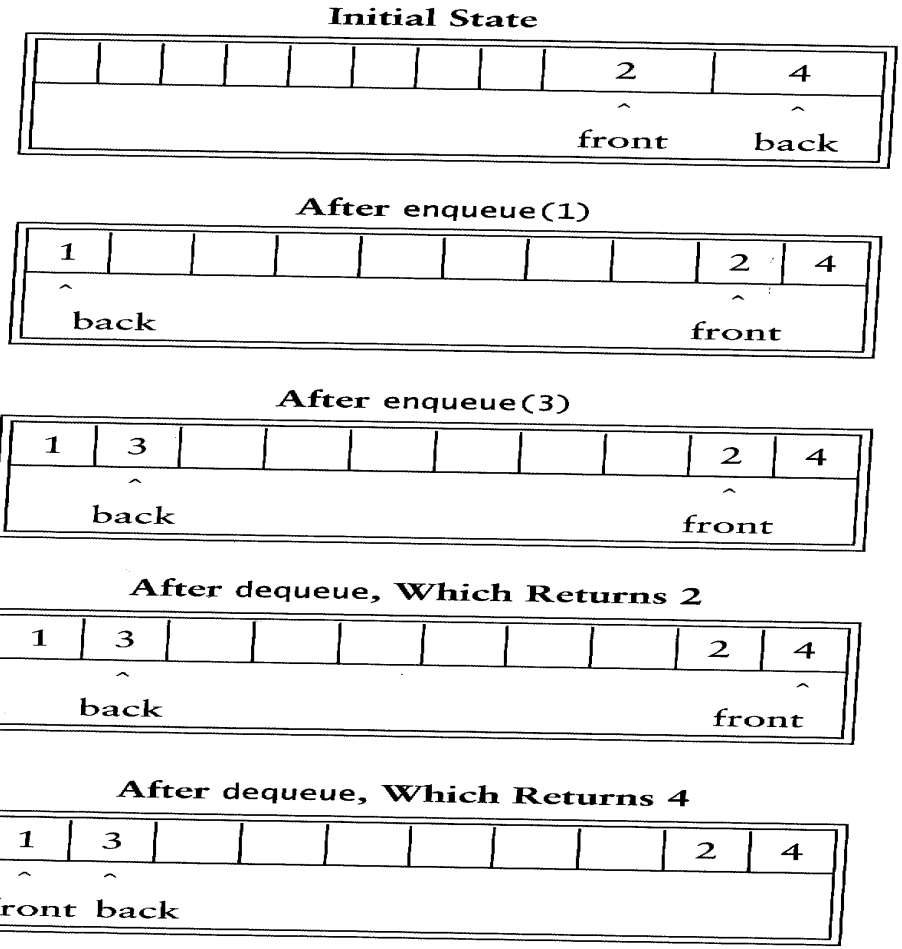
```
        /**
         * Print List from ListNode p onwards; assume friendship granted.
         */
        template <class Object>
        void printList( ListNode<Object> *p )
        {
/*1*/       if( p == NULL )
/*2*/           return;

/*3*/       cout << p->element << endl;
/*4*/       printList( p->next );
        }
```

подобрена версия на предишната реализация на printList():

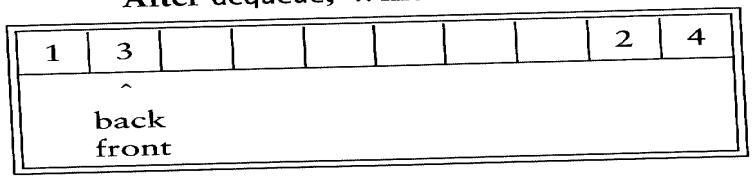
```
/**
 * Print List from ListNode p onwards; assume friendship granted.
 */
template <class Object>
void printList( ListNode<Object> *p )
{
  while( true )
  {
    if( p == NULL )
        return;

    cout << p->element << endl;
    p = p->next;
  }
}
```

опашки

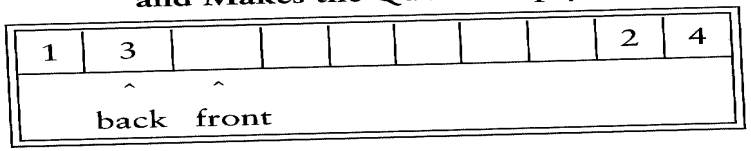
реализация на опашки чрез списък;
реализация на опашка чрез масив:



After dequeue, Which Returns 1



After dequeue, Which Returns 3 and Makes the Queue Empty



декларация на клас – опашка . (реализация през масив)

```
/**
 * Construct the queue.
 */
template <class Object>
Queue<Object>::Queue( int capacity ) : theArray( capacity )
{
    makeEmpty( );
}

/**
 * Make the queue logically empty.
 */
template <class Object>
void Queue<Object>::makeEmpty( )
{
    currentSize = 0;
    front = 0;
    back = -1;
}
```

реализация на методи на опашка (през масив) :

```
/**
 * Construct the queue.
 */
template <class Object>
Queue<Object>::Queue( int capacity ) : theArray( capacity )
{
    makeEmpty( );
}

/**
 * Make the queue logically empty.
 */
template <class Object>
void Queue<Object>::makeEmpty( )
{
    currentSize = 0;
    front = 0;
    back = -1;
}
```

реализация на методи на опашка (през масив) :

```
/**
 * Insert x into the queue.
 * Throw Overflow if the queue is full.
 */
template <class Object>
void Queue<Object>::enqueue( const Object & x )
{
    if( isFull( ) )
        throw Overflow( );
    increment( back );
    theArray[ back ] = x;
    currentSize++;
}
```

```
/**
 * Internal method to increment x with wraparound.
 */
template <class Object>
void Queue<Object>::increment( int & x )
{
    if( ++x == theArray.size( ) )
        x = 0;
}
```

реализация на методи на опашка (през масив) :

```
/**
 * Return and remove the least recently inserted item from the queu
 * Throw Underflow if the queue is empty.
 */
template <class Object>
Object Queue<Object>::dequeue( )
{
    if( isEmpty( ) )
        throw Underflow( );

    currentSize--;
    Object frontItem = theArray[ front ];
    increment( front );
    return frontItem;
}
```

използване на опашки:

- **в реализация на графи;**
- **опашки задачи напр. към принтер;**
- **опашки задачи към file server;**
- **опашки от заявки за обслужване;**
- **опашки от клиенти;**
- **опашки от съобщения в Windows;**

Hash tables – a bit of theory

-A hash table is an effective data structure for implementing dictionaries and operations on elements

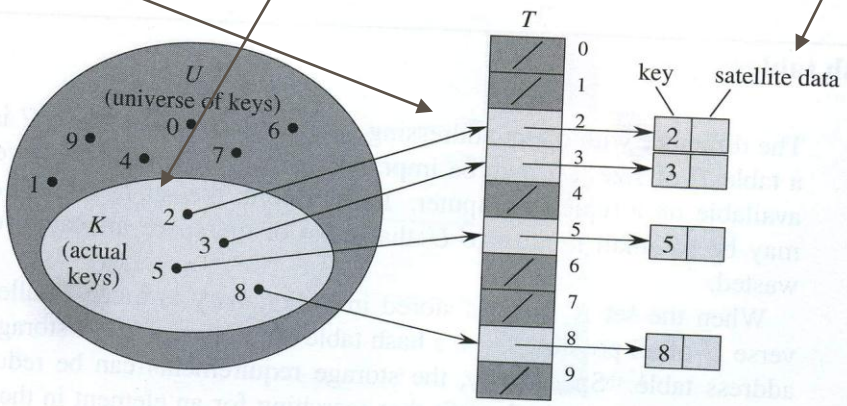
-Generalization of the simpler notion of an ordinary array with direct addressing (time of O(1))
Direct addressing is applicable when 1 position ← →1 key

- instead of using key as an index into the array directly, the index can be computed from the key

DIRECT-ADDRESS TABLES – basic idea

When the universe 'U' of keys is reasonably small;

To represent a dynamic set of data we use an array (direct-address table : T(0… m-1)) in which each position (slot) corresponds to a key in U. Keys are stored with the satellite data. Operations should run for O(1) time



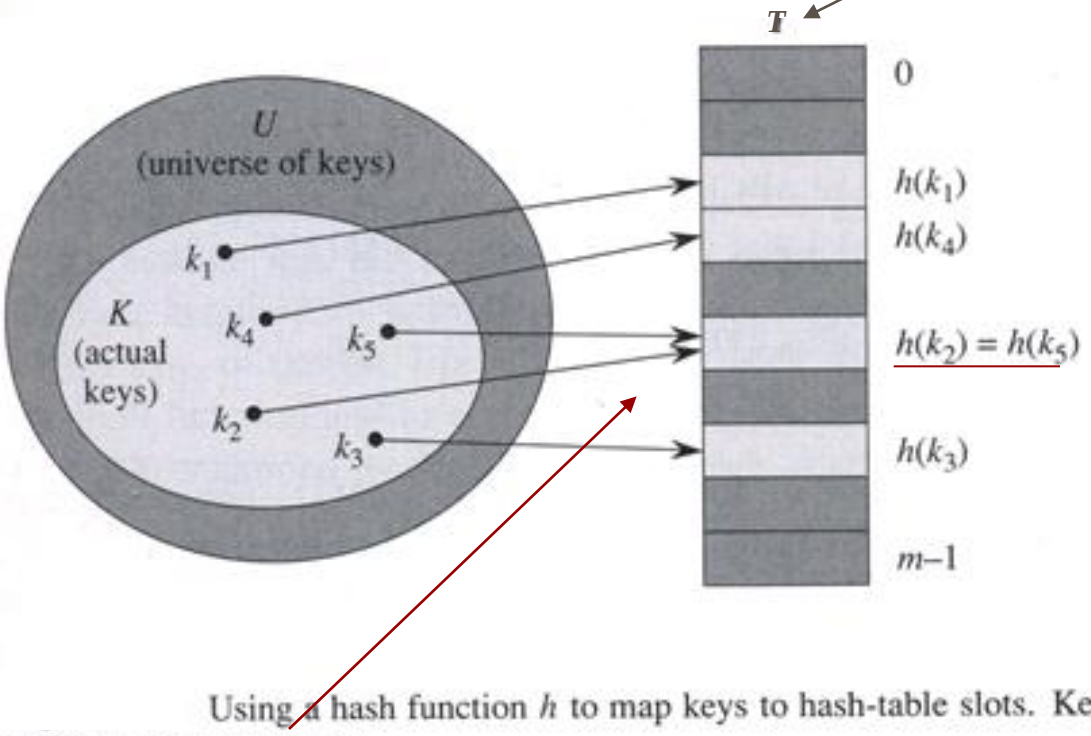
Implementing a dynamic set by a direct-address table T. Each key in the universe $U = \{0, 1, \ldots, 9\}$ corresponds to an index in the table. The set $K = \{2, 3, 5, 8\}$ of actual keys determines the slots in the table that contain pointers to elements. The other slots, heavily shaded, contain NIL.

Table 'T' now is called - hash table

-When U is too large and the set of used keys (K) is smaller, we have to waste time & memory
Is better to use not direct-addressable table , but a hash with time O(1) again.

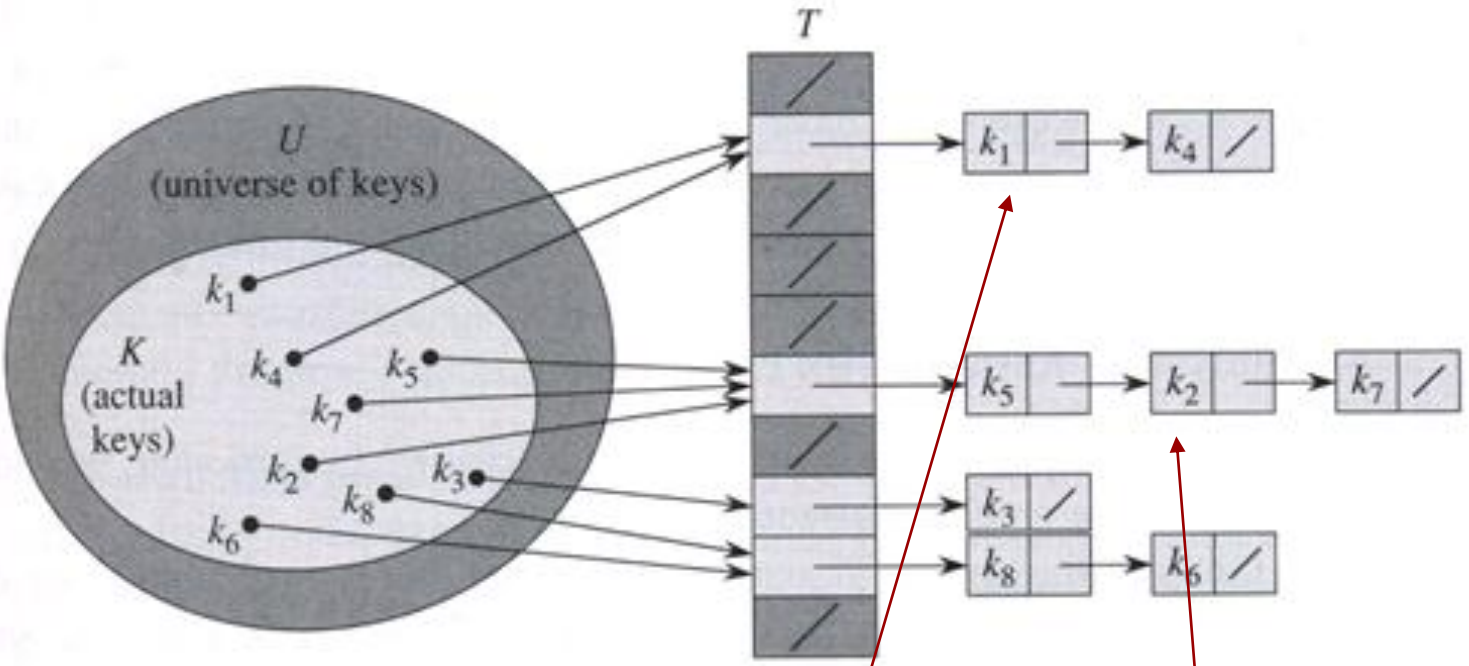
-Use of hash function ***h(k)*** *to compute the slot from the key 'k' (before element with key k had been stored in slot k)*

- 'h' maps U keys into the slots of a hash table T[0… m-1]



Using a hash function h to map keys to hash-table slots. Keys k_2 and k_5 map to the same slot, so they collide.

Collision resolution by chaining



Collision resolution by chaining. Each hash-table slot $T[j]$ contains a linked list of all the keys whose hash value is j. For example, $h(k_1) = h(k_4)$ and $h(k_5) = h(k_2) = h(k_7)$.

Hash functions

How to find a good hash function?

Uniform hashing

for example if keys are random real numbers '<u>k</u>' independently and uniformly distributed in the range 0<= k <= 1, the hash function:

$$h(k) = \lfloor km \rfloor$$

is a *simple uniform hash function* (*'m" is number of slots in 'T')*

The selection of the hash function depends of used data.

Example:

compiler's symbol table for storing identifiers from the program. Closely related symbols like *pt* and *pts* may occur frequently . A good hash function would minimize the chance that such variant hash to the same slot

Hash functions by division method

Gives good results, because the prime number (the divider) is easy to be chosen to be unrelated to any pattern in the distribution of keys.

- Hash function is created by taking the remainder of 'k' divided by 'm':

$$h(k) = k \bmod m$$

Example 1: let the hash table has size of m = 12 (*we have chosen divider 'm'*)
the current key is: k = 100
so: h(k) = 4

Some values of 'm' must be avoided – 'm' should not be a power of 2
if $m=2^p$, h(k) will be just the low order p-bits of k. They probably will be same for most of the time.

Example 2: having hash table with n=2000 strings; we want:
it's satisfactory to look up to 3 elements in collision, so let the hash table be of size m = 701
(701 is near to 2000/3; not a power of 2)
hash function for this may be:

$$h(k) = k \bmod 701$$

Hash functions through multiplication method

- 2 steps are to be executed:
  - Multiply the key 'k' by a constant A in range $0 < A < 1$ and extract the fractional part of 'kA'
  - Then multiply this value by 'm' and take the floor of the result

  In short the hash function is:

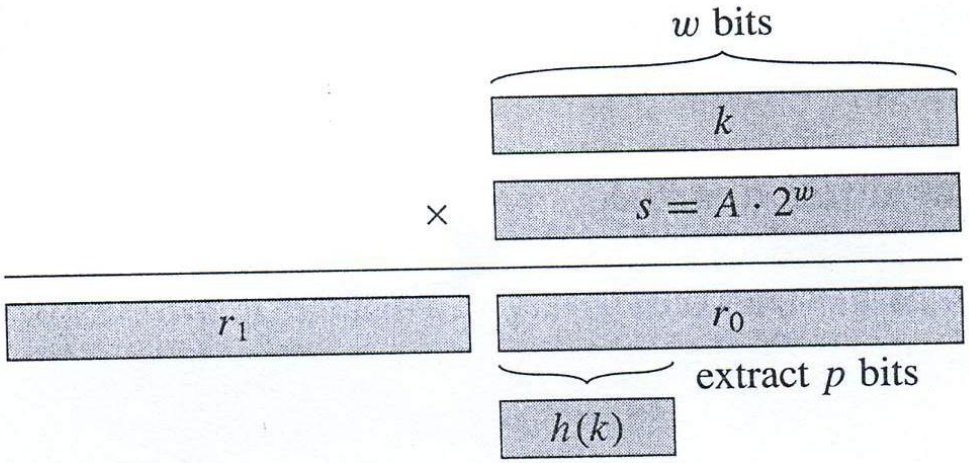
$$h(k) = \lfloor m\,(kA \bmod 1) \rfloor$$

Where kA mod 1 means the fractional part of kA: $kA - \lfloor kA \rfloor$

- Now m may be power of 2 ($m = 2^p$, where 'p' is some integer)
- Suppose w – word size;
  'k' fits into single word
  'A' may be $= s/2^w$, where 's' is integer in the range $0 < s < 2^w$

The following chart describes the process:

Hash functions – the multiplication method



The multiplication method of hashing. The w-bit representation of the key k is multiplied by the w-bit value $s = A \cdot 2^w$. The p highest-order bits of the lower w-bit half of the product form the desired hash value $h(k)$.

•Different 'A' works differently. The optimal choice depends on the characteristics of the data being hashed. Knuth suggests that:

A around 0.6180339887… and is likely to work reasonably well

Example: let k=123456, p=14, m= 2^{14} = 16384 and w = 32.

'A' is chosen to be fraction of the form s/2^{32}, that is closest to 0,61803398.. , so that A = 2654435769/2^{32} . Then k*s = 327706022297664 = 76300 * 2^{32} + 17612864

so r_1 = 76300 and r_0 = 17612864. The 14 most significant bits of r0 yield the value of h(k) = 67

Hash functions – universal hashing

- ***Universal hashing*** – when hash function is chosen ***randomly*** in a way that is ***independent*** of the keys that are actually going to be stored. Then will be impossible to chain all data into 1 slot
- Usually a *collection of functions* must be ready and ‘universal hashing ‘ means to choose one of them at random

Open addressing

- All elements are stored in the hash table itself (the key ‘k’ is identical to the element for key ‘k’)
- No lists, no elements are outside the table, no chaining
- No pointers are used. Instead we examine sequence of slots
- The table includes slots or empty elements

1. Open addressing – linear probing (successively examining)

- Given an ordinary hash function h’: U →{0,1,…m-1} which we refer to as an auxiliary hash function, the method uses the hash function:

$$h(k,i) = (h'(k) + i) \bmod m$$

for i = 0,1,2,…, m-1

There are only ‘m’ distinct probe sequences.

Linear probing is easy to implement, but suffers from ***primary clustering*** – needs long runs on occupied slots

2.Open addressing – quadratic probing

- Uses hash function:

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

where **h'** is auxiliary hash function; c_1 and c_2 are auxiliary constants; $i = 0,1,\ldots m-1$

- Works with jumps, depending of i and 'c'
- Works better than linear probing, but empty slots exists
- If 2 keys have the same initial probe position, there probe sequence are the same (*secondary clustering*)

3. Open addressing – double hashing

- The best method between open-addressing methods – near too the randomly chosen permutations
- Uses a hash function:

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

Where h_1 and h_2 are auxiliary hash functions

- $h_2(k)$ must be relatively prime to the hash table size 'm' for the entire hash table to be searched
- Usually 'm' is a power of 2 and h_2 is designed to always produce an odd number. Or:

'm' be prime and

$h_1(k) = k \bmod m$

$h_2(k) = 1 + (k \bmod m')$

where m' is slightly less than m (for example m-1).

Example: k= 123456;

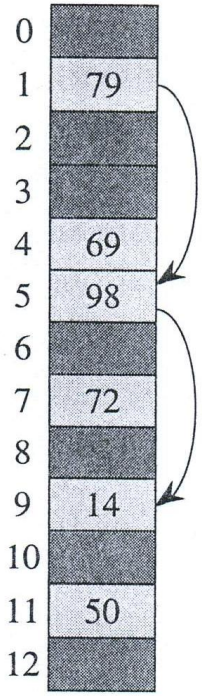
m = 701; m' = 700 → $h_1(k) = 80$

$h_2(k) = 257$

so the first probe is to position 80 and then every slot (modulo m) is examined until the key is found, or every slot is examined

257^{th}

Open addressing – double hashing



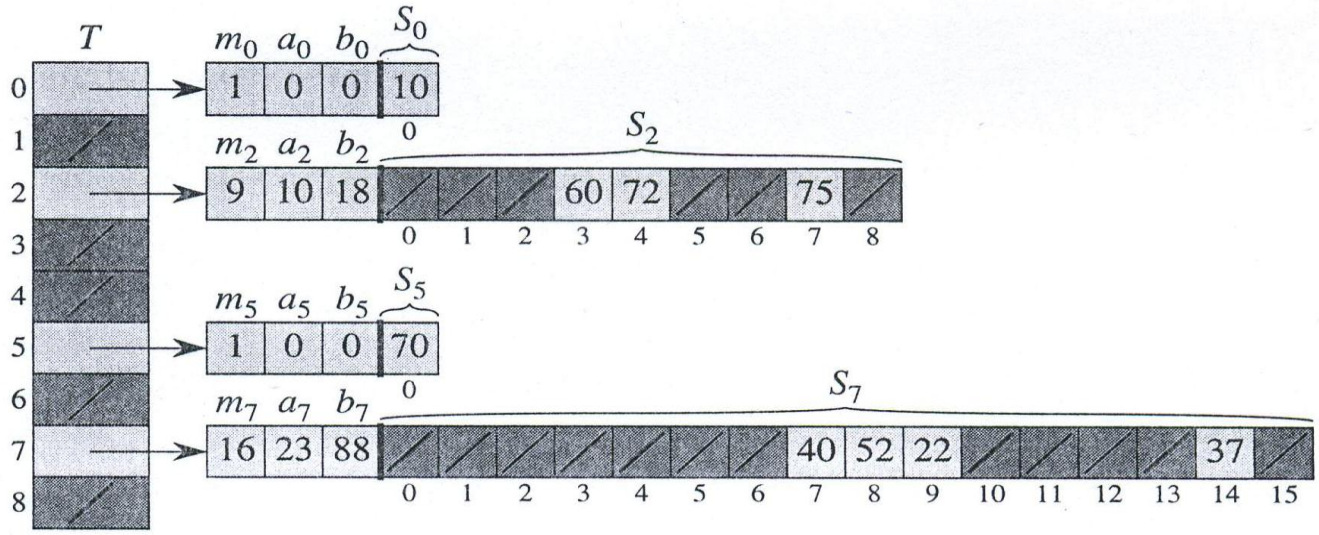
Insertion by double hashing. Here we have a hash table of size 13 with $h_1(k) = k \bmod 13$ and $h_2(k) = 1 + (k \bmod 11)$. Since $14 \equiv 1 \pmod{13}$ and $14 \equiv 3 \pmod{11}$, the key 14 is inserted into empty slot 9, after slots 1 and 5 are examined and found to be occupied.

Perfect hashing

- **Hashing can be used to obtain *worst-case* performance when the set of keys is static – stored once into a table and never changed. Example – the set of reserved words in a programming language, or set of file names on a CD-ROM. We call *perfect hashing if the worst-case number of memory accesses required to perform a search is O(1)***

- **The basic idea of perfect hashing is to use a 2-levels hashing scheme with universal hashing at each level:**

Perfect hashing



Using perfect hashing to store the set $K = \{10, 22, 37, 40, 52, 60, 70, 72, 75\}$. The outer hash function is $h(k) = ((ak + b) \bmod p) \bmod m$, where $a = 3$, $b = 42$, $p = 101$, and $m = 9$. For example, $h(75) = 2$, so key 75 hashes to slot 2 of table T. A secondary hash table S_j stores all keys hashing to slot j. The size of hash table S_j is m_j, and the associated hash function is $h_j(k) = ((a_j k + b_j) \bmod p) \bmod m_j$. Since $h_2(75) = 7$, key 75 is stored in slot 7 of secondary hash table S_2. There are no collisions in any of the secondary hash tables, and so searching takes constant time in the worst case.

•The first level is the same as for hashing with chaining: n keys are hashed into m slots using hash function h from a family of universal hash functions.

•Instead of making a list of the keys hashing to slot j, we use a small secondary hash table Sj With an associated hash functions hj and no collisions at the secondary level (the size m of Sj must be not exceedingly big