# Chap. 3 Lists, Stacks, and Queues

## 3.1 Abstract Data Types (ADTs)

- An ADT is a set of objects (of the same type) with a set of operations
- ADT can be naturally implemented by C++ classes
- Conceptually, a data structure (list, set, graph) is an ADT

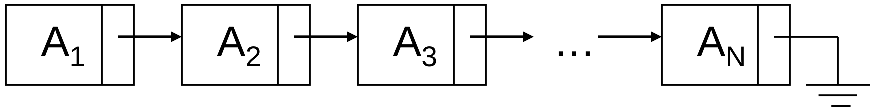
## 3.2 Lists

- List ADT
  - A sequence of objects $A_1, A_2, \ldots, A_N$
  - Common list operations include: *printList, makeEmpty, find, insert, remove, findKth, next, findPrevious*, etc.
- Array implementation of lists
  - Size estimate
  - *printList & find*: linear time O(N)
  - *findKth*: constant time O(1)

| | | | | |
|---|---|---|---|---|
| $A_1$ | $A_2$ | $A_3$ | ………….. | $A_N$ |

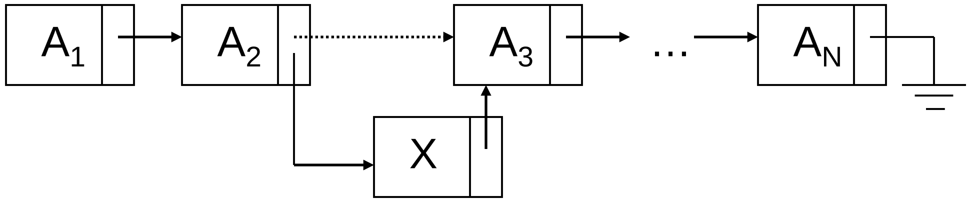
- *insert & remove*: involving memory management, linear cost.
- A generally inefficient implementation for dynamic lists, but good for relatively static lists.

– Linked lists

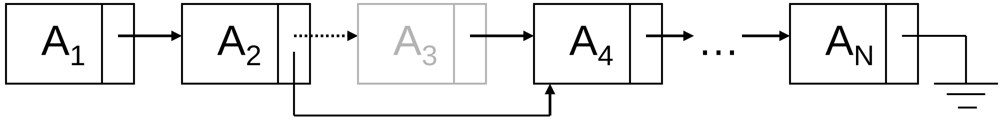



The nodes are not necessarily contiguous memory cells.

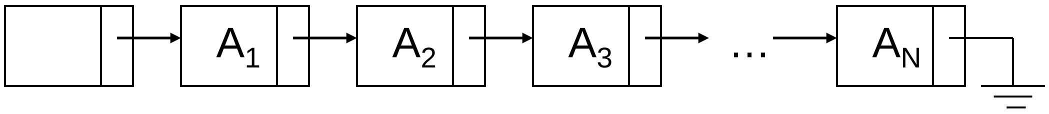
- *printList, findKth*: linear time
- *insert*: constant time



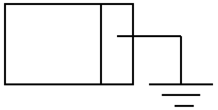

- *remove*: constant time



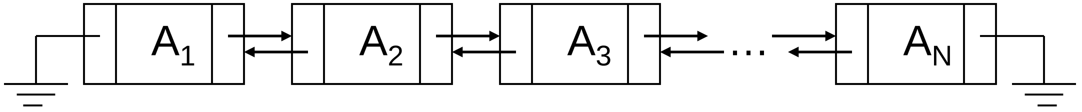
- Header
  - A dummy node pointing to the first node of the list




  - Can reprint empty list

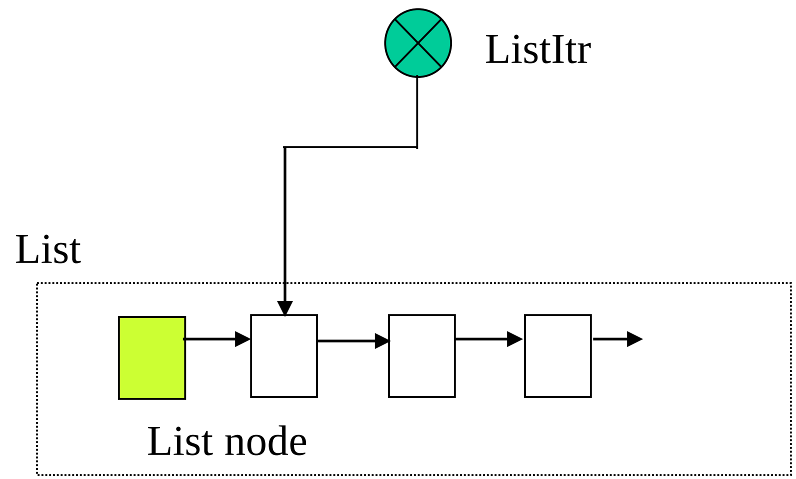


  - Can avoid failures in special cases for list operations
  - C++ implementations `(Textbook, page 73-79)`
- Doubly linked lists




  - More memory
  - Simplifies deletion (*findPrevious*)

```
#ifndef _LinkedList_H
#define _LinkedList_H
#include "dsexceptions.h"
#include <iostream.h>    // For NULL
// List class CONSTRUCTION: with no initializer
// Access is via ListItr class
// boolean isEmpty( )
//                        --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ListItr zeroth( )
//                        --> Return position to prior to first
// ListItr first( )       --> Return first position
// void insert( x, p )
//              --> Insert x after current iterator position p
// void remove( x )       --> Remove x
// ListItr find( x )      --> Return position that views x
// ListItr findPrevious( x )
//                      --> Return position prior to x
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>;
    };
```

```
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;
      };
 // ListItr class; maintains "current position" CONSTRUCTION:
Package friendly only, with a ListNode
```

```
// ******************PUBLIC OPERATIONS*********
    // bool isPastEnd( )     --> True if past end position in list
    // void advance( )       --> Advance (if not already null)
    // Object retrieve       --> Return item in current position
    template <class Object>
    class ListItr
    {
      public:
        ListItr( ) : current( NULL ) { }
        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
    };
    #include "LinkedList.cpp"
    #endif
```

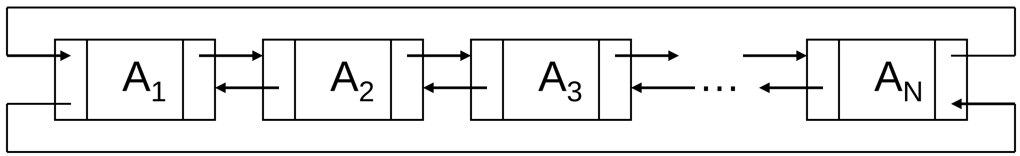
```
#include "LinkedList.h"
/* Construct the list.   */
template <class Object>
List<Object>::List( )
{           header = new ListNode<Object>;
}
/* Copy constructor.   */
template <class Object>
List<Object>::List( const List<Object> & rhs )
{
   header = new ListNode<Object>;
   *this = rhs;
}
/* Destructor.        */
template <class Object>
List<Object>::~List( )
{
   makeEmpty( );
   delete header;
}
```

```
/* 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;
      }
      /* Make the list logically empty. */
      template <class Object>
      void List<Object>::makeEmpty( )
      {           while( !isEmpty( ) )
            remove( first( ).retrieve( ) );
      }
      /* Return an iterator representing the header node. */
      template <class Object>
      ListItr<Object> List<Object>::zeroth( ) const
      {           return ListItr<Object>( header );
      }
/* 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 );
      }
```

```
/* 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 );
      }
/* 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 );
      }
/* 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 );
      }
```

```
/* 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;
     }
  }
  /* Deep copy of linked lists.*/
template <class Object>
const List<Object>&List<Object>::operator=( const List<Object> & rhs )
 {   if( this != &rhs )
     {
        makeEmpty( );
        ListItr<Object> ritr = rhs.first( );
        ListItr<Object> itr = zeroth( );
        for( ; !ritr.isPastEnd( ); ritr.advance( ), itr.advance( ) )
           insert( ritr.retrieve( ), itr );
     }
     return *this;
 }
```

– Circular linked lists



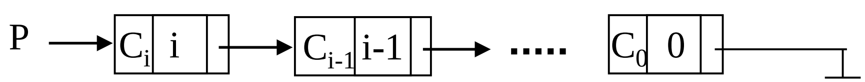

- No need for header
- No special case for "*next*" & *"findPrevious*"

– Examples

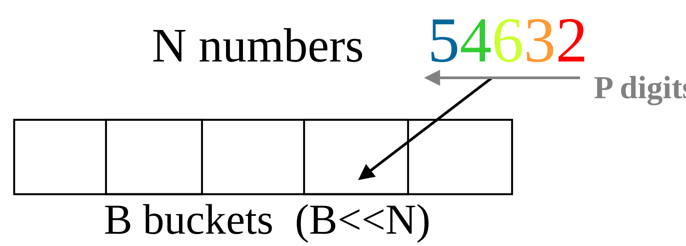
- Polynomials
`(Textbook, page 81-83)`

$$F(x) = \sum_{i=0}^{N} c_i x^i$$

$$c_n x^n + c_{n-1} x^{n-1} + \ldots . + c_1 x + c_0$$

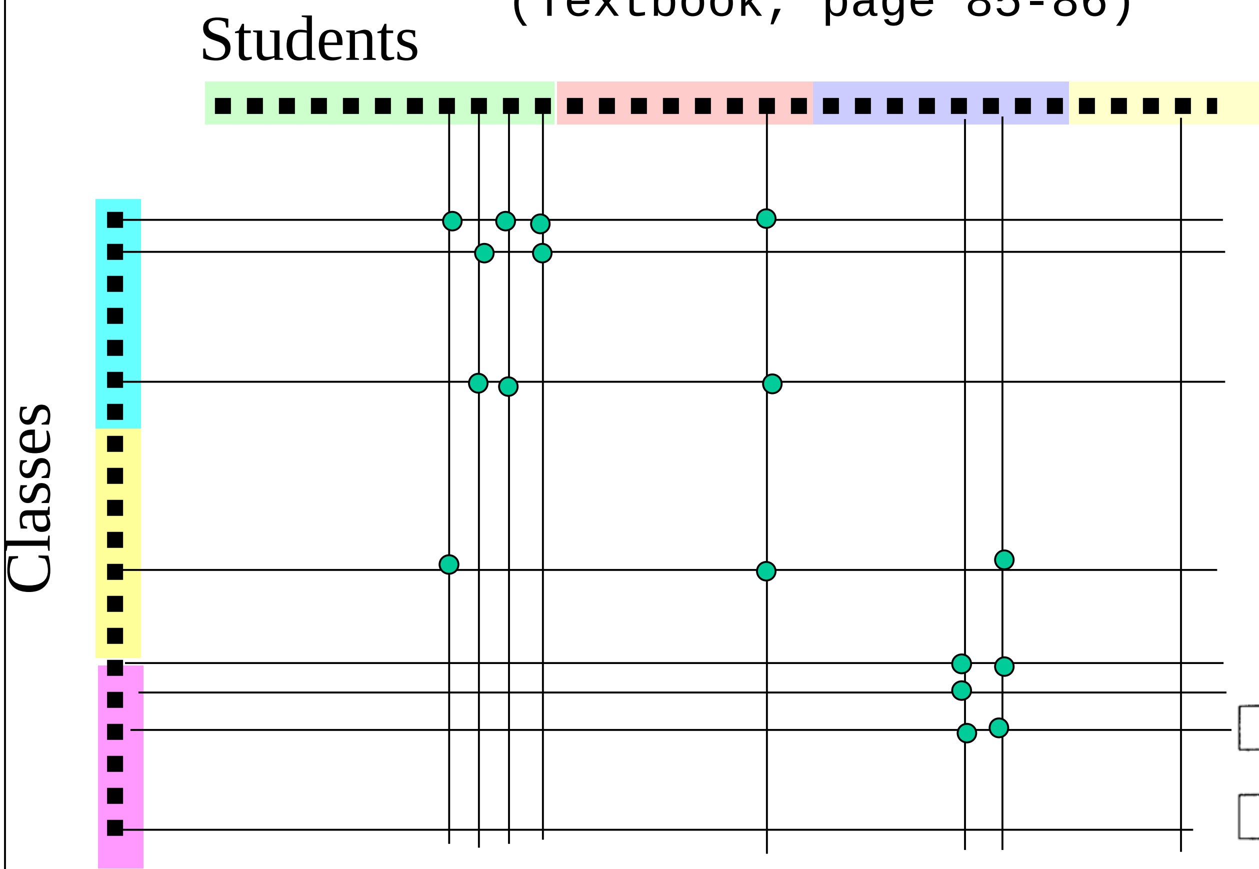



- Radix Sort - O(P(N+B))
`(Textbook, page 83-85)`



•Multi-lists

(Textbook, page 85-86)



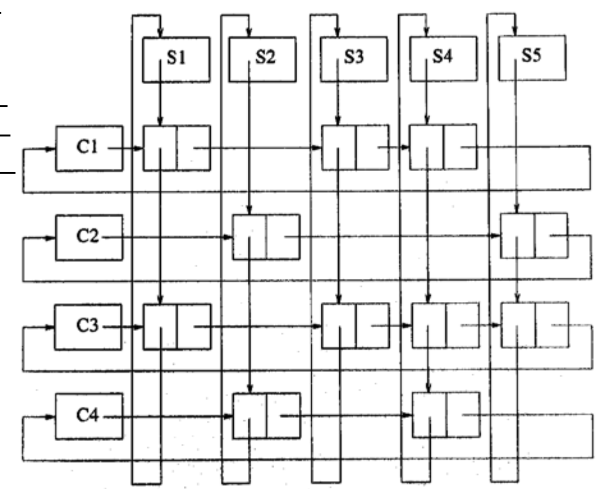




Figure 3.28 Multilist implementation for registration problem

$$F(x) = \sum_{i=0}^{n} a_i x^i \qquad G(x) = \sum_{i=0}^{m} b_i x^i$$

$$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \qquad b_m x^m + b_{m-1} x^{m-1} + \ldots\ldots\ldots + b_1 x + b_0$$

$$F(x) + G(x), \quad F(x)^* \, G(x)$$

```
/* This code doesn't really do much, and abstraction is not built in.
 * Thus, I haven't bothered testing it exhaustively.*/
  #include <iostream.h>
  #include "vector.h"

  class Polynomial
  {       enum { MAX_DEGREE = 100 };
      friend int main( );   // So I can do a quick test.
    public:
      Polynomial( );
      void zeroPolynomial( );
      Polynomial operator+( const Polynomial & rhs ) const;
      Polynomial operator*( const Polynomial & rhs ) const;
      void print( ostream & out ) const;
    private:
      vector<int> coeffArray;
      int highPower;
  };
```

```
int max( int a, int b )
{           return a > b ? a : b;
}

Polynomial::Polynomial( ) : coeffArray( MAX_DEGREE + 1 )
{          zeroPolynomial( );
}

void Polynomial::zeroPolynomial( )
{          for( int i = 0; i <= MAX_DEGREE; i++ )
        coeffArray[ i ] = 0;
    highPower = 0;
}
```

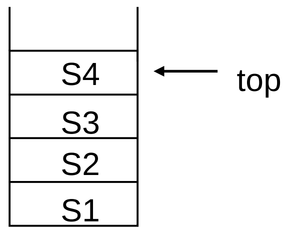
```
Polynomial Polynomial::operator+( const Polynomial & rhs )
const
      {  Polynomial sum;
         sum.highPower = max( highPower, rhs.highPower );
         for( int i = sum.highPower; i >= 0; i-- )
            sum.coeffArray[ i ] = coeffArray[ i ] + rhs.coeffArray[ i ];
         return sum;
      }

Polynomial Polynomial::operator*( const Polynomial & rhs ) const
      {          Polynomial product;
         product.highPower = highPower + rhs.highPower;
         if( product.highPower > MAX_DEGREE )
            cerr << "operator* exceeded MAX_DEGREE" << endl;
         for( int i = 0; i <= highPower; i++ )
            for( int j = 0; j <= rhs.highPower; j++ )
               product.coeffArray[ i + j ] +=
                     coeffArray[ i ] * rhs.coeffArray[ j ];
         return product;
      }
void Polynomial::print( ostream & out ) const
      {    for( int i = highPower; i > 0; i-- )
            out << coeffArray[ i ] << "x^" << i << " + ";
         out << coeffArray[ 0 ] << endl;
      }
```

```
ostream & operator<<( ostream & out, const
Polynomial & rhs )
      {   rhs.print( out );
          return out;
      }


int main( )
      {   Polynomial p;
          Polynomial q;

          p.highPower = 1;
          p.coeffArray[ 0 ] = 1;
          p.coeffArray[ 1 ] = 1;

          q = p + p;
          p = q * q;
          q = p + p;
          cout << q << endl;
          return 0;
      }
```

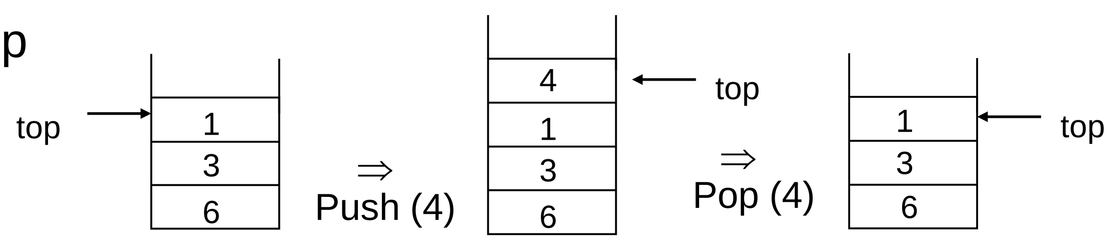
# 3.3 Stacks

– Stack model

- A stack is a list with the restriction that insertion & deletion can be performed only at the end (or top) of the list.

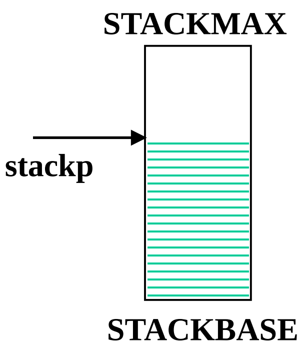



- Only the top node is accessible
- Last in, first out (LIFO)
- Push
- Pop




- A stack can be empty, “pop” from an empty stack is an error
- A stack can never be full (assuming infinite memory)

- Implementation by linked lists `(Fig. 3.41, page 94)`
  - Methods implementation `(Fig. 3.42 - 3.47)`
- Implementation by array `(Fig. 3.48, page 99)`
  - Need to set the maximum stack size
  - Pop & push: constant time (fast)
  - More commonly used than list implementation




```
PUSH:   if(stackp)>STACKMAX   then
{
        (stackp)+1 → stackp
        x → (stackp)
}

POP:   if (stackp)<STACKBASE then
{
        ((stackp)) → x;
        (stackp)-1 → stackp
}
```

Push and Pop in C or Assembly

```
// Stack class -- array implementation
   // CONSTRUCTION: with or without a capacity; default is 10
// ******************PUBLIC OPERATIONS***********
      // void push( x )         --> Insert x
      // void pop( )            --> Remove most recently inserted item
      // Object top( )          --> Return most recently inserted item
      // Object topAndPop( )    --> Return and remove most
       //    recently inserted item
      // bool isEmpty( )        --> Return true if empty; else false
      // bool isFull( )         --> Return true if full; else false
      // void makeEmpty( )      --> Remove all items
      // ******************ERRORS********************
      // Overflow and Underflow thrown as needed
```

```
#ifndef _STACKAR_H
#define _STACKAR_H
#include "vector.h"
#include "dsexceptions.h"
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;
};
#include "StackAr.cpp"
#endif
```

```
#include "StackAr.h"
/**Construct the stack. */
    template <class Object>
    Stack<Object>::Stack( int capacity ) : theArray( capacity )
    {        topOfStack = -1;
    }
/*** Test if the stack is logically empty.   Return true if empty,
false, otherwise. */
    template <class Object>
    bool Stack<Object>::isEmpty( ) const
    {        return topOfStack == -1;
    }
/**Test if the stack is logically full. Return true if full, false
otherwise.        */
    template <class Object>
    bool Stack<Object>::isFull( ) const
    {        return topOfStack == theArray.size( ) - 1;       }
    /**        * Make the stack logically empty.        */
    template <class Object>
    void Stack<Object>::makeEmpty( )
    {        topOfStack = -1;       }
/**Get the most recently inserted item in the stack.  Does not
alter the stack. Return the most recently inserted item in the
stack. Exception Underflow if stack is already empty. */
    template <class Object>
    const Object & Stack<Object>::top( ) const
    {     if( isEmpty( ) )
throw Underflow( );
        return theArray[ topOfStack ];
    }
/* * Remove the most recently inserted item from the
stack.Exception Underflow if stack is already empty.        */
    template <class Object>
    void Stack<Object>::pop( )
    {        if( isEmpty( ) )
          throw new Underflow( );
        topOfStack--;
    }
/**Insert x into the stack, if not already full.Exception
Overflow if stack is already full.*/
    template <class Object>
    void Stack<Object>::push( const Object & x )
    {        if( isFull( ) )
          throw Overflow( );
        theArray[ ++topOfStack ] = x;
    }
/**Return and remove most recently inserted item from the
stack.Return most recently inserted item.Exception
Underflow if stack is already empty.        */
    template <class Object>
    Object Stack<Object>::topAndPop( )
    {        if( isEmpty( ) )
          throw Underflow( );
        return theArray[ topOfStack-- ];
    }
```

– Stack applications

- Balancing symbols: linear time `(page 101 - 102)`

```
[xxx(xxx)xx(x)x]xxxx{xx[x(xxxx)x(x)]xxx}
    Read characters until end-of-flie
    Push opening symbol to stack,
    pop counterpart if closing symbol is read
```

- Postfix expressions ( inverse Polish notation) `(page 104)`

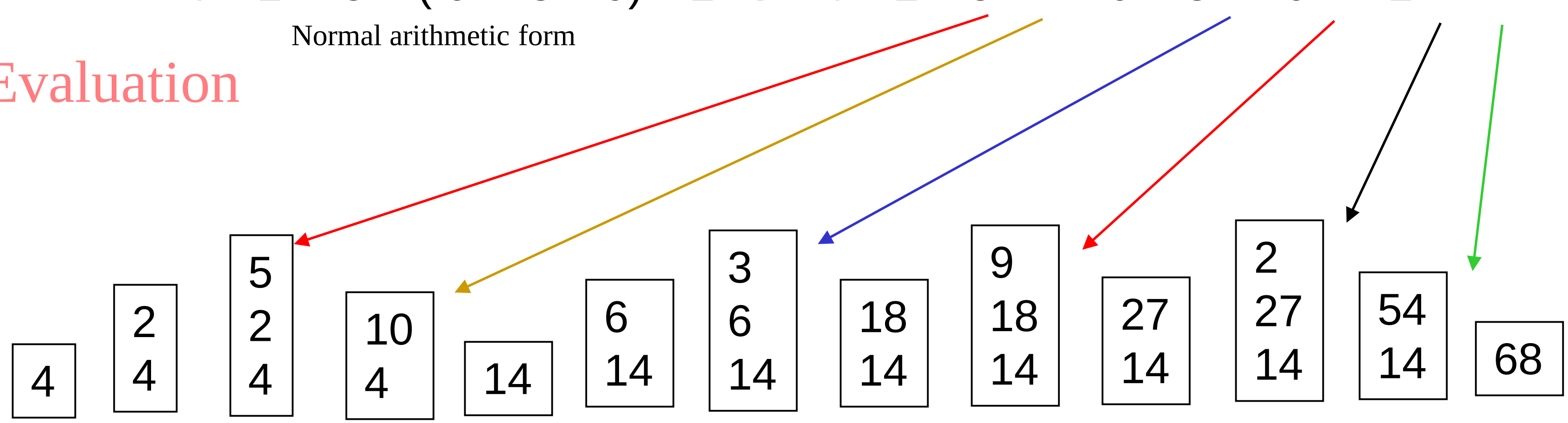
$$a*b+c+d*e \Rightarrow ab*c+de*+$$

Polish notation

4 + 2 * 5 + (6 * 3 + 9) * 2 → 4 2 5 * + 6 3 * 9 + 2 * +

Normal arithmetic form

Evaluation



- Infix to Postfix Conversion `(page 106)`

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

# Conversion

Operator precedence: ( , * /, + -, )

**If '(' then push**

**else if ')' then pop entries until '('**

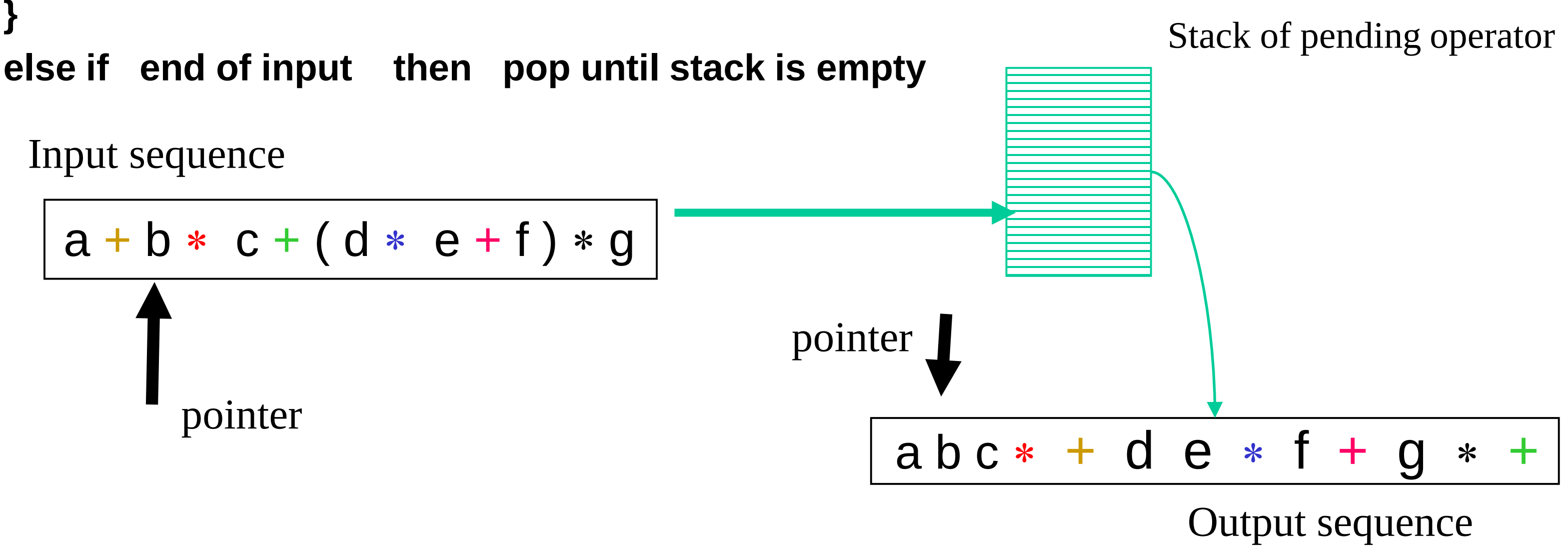
**else if 'operator' then {**

**pop operators with higher or same precedence**

**push**

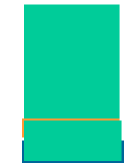
**}**

**else if end of input then pop until stack is empty**

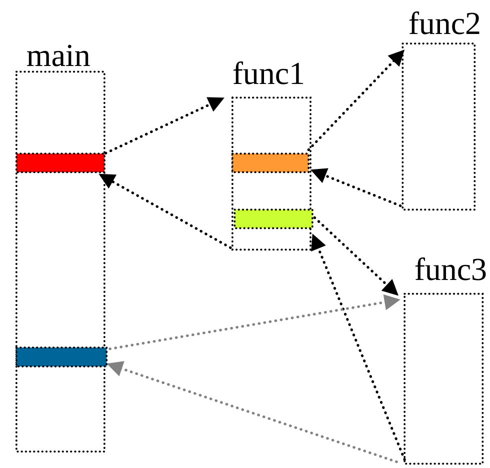


## –Function calls using stacks

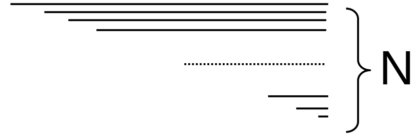
Stack for activation record



- •Saving local variables using stack
- •Recursion: stack implementation
- •Stack overflow with runaway recursion



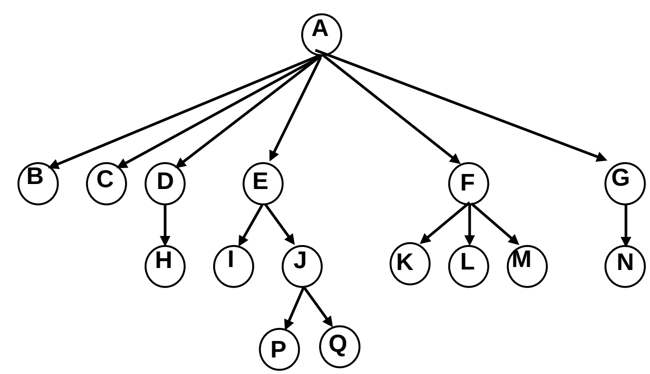

tail recursion 3455433223454




Tail recursion: N elements → N layers of recursion

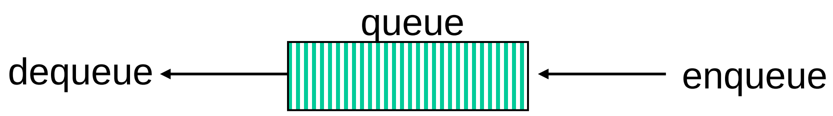
Tree: N elements → log N layers of recursion

example: `fig.3.55`



# 3.4 Queues

– Queue Model
  - queue is a list, with insertion done only at one end and deletion done at the other end.
  - enqueue: insert an element at the end of the queue
  - dequeue: delete (and return) the element at the start of the queue
  - first in first out model




– Linked list implementation of queues
  - operating as a list
  - constant time for enqueue & dequeue (keeping pointer to both the head and tail of the list)

- Array implementation of queues
  - front pointer, back pointer, current size
  - circular array `(Fig. in page 111)`
  - C++ implementation `(Fig. 3.58 - 3.61)`
- Applications of queues
  - printer job queues
  - telephone queues
  - class waiting list