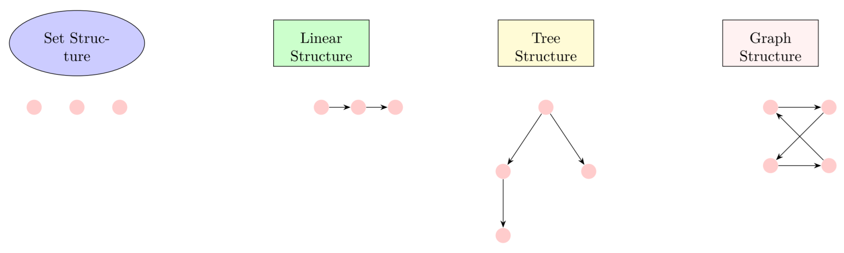
oiWiki

1.
2.
3.

-
  .
-
  .
-
  .
-


Set Structure
Linear Structure
Tree Structure
Graph Structure

-

-
-
-
-
-
-
- **traverse**
-

-
-
-
  -
  -
  -

-
-
-
-

```
template <class  >
class
{
    //
};
```

```
class
{
  [private:]
    //
  public:
    //
};
```

- virtual
-

  ```
  virtual      ( )
  {
    //
  };
  ```

-
-
-

-
-

  ```
  virtual      ( ) = 0;
  ```

-

-
-

-
-
-

-

  ```
  class    : [  ]
  {
    //
  };
  ```

-

  - `public`
  - `protected`
  - `private`

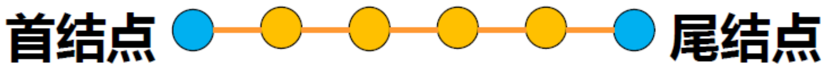
- `class` `private` `struct` `public`

	`public`	`protected`	`private`
`public`	public	protected	private
`protected`	protected	protected	private
`private`			

`const` `int search(constelemType&x) const`

1. `x` `const` `&`
   - `const` `x` `search` `x`
   - `const` `x`
   - `x` **`const`**
   - `&` `x` `x`
2. `search` `const`
   - `const`
   - `const` `const`
     `const`

-
-
-



- **create()**
- **clear()**
- **length()**
- **insert(i,x)** x
- **remove(i)**
- **search(x)** x
- **visit(i)**
- **traverse()**

```
template <class elemType>
class list
{
  public:
    virtual void clear() = 0;                              //
    virtual int length() const = 0;                        //
    virtual void insert(int i, const elemType &x) = 0;  //
    virtual void remove(int i) = 0;                        //
    virtual int search(const elemType &x) const = 0;    //
    virtual elemType visit(int i) const = 0;            //
    virtual void traverse() const = 0;                  //
    virtual ~list() {}                                     //
};
```

-
-

-
-
-
-

```
template <class elemType>
class seqList : public list<elemtype>
{
  private:
    elemType *data;
    int currentLength;
    int maxSize;
    void doubleSpace();
  public:
    seqList(int initSize = 10);             //
    ~seqList()                              //
    {
      delete[] data;
    }
    void clear()                            //
    {
      currentLength = 0;
    }
    int length() const                      //
    {
      return currentLength;
    }
    void insert(int i, const elemType &x);  //
    void remove(int i);                     //
    int search(const elemType &x) const;    //
    elemType visit(int i) const;            //
    void traverse() const;                  //
};
```

-
-
-
-

```
template<class elemType>
seqList<elemType>::seqList(int initSize)
{
  data = new elemType[initSize];
  maxSize = initSize;
  currentLength = 0;
}
```

seqList

-
- ~
-
-

-
-

```
~seqList()
{
  delete[] data;
}  //
```

search

0 x

```
template<class elemType>
int seqList<elemType>::search(const elemType &x) const
{
  int i ;
  for (i = 0 ; i < currentLength && data[i] != x ; ++i);
  if (i == currentLength)
  {
    return -1;
  }
  else
  {
    return i;
  }
}
```

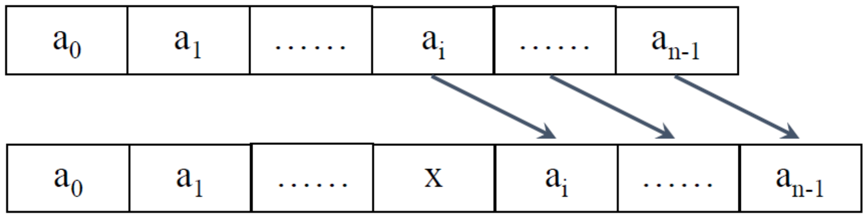
traverse

0

```
template<class elemtype>
void seqList<elemType>::traverse() const
{
  for (int i = 0 ; i < currentLength ; ++i)
  {
    cout << data[i] << ' ';
  }
}
```

insert

- i x
- maxSize maxSize



```
template<class elemType>
void seqList<elemType>::insert(int i , const elemType &x)
{
  if (currentSize = maxSize)
  {
    doubleSpace();
  }
  for (int j = currentLength ; j > i ; --j)
  {
    data[j] = data[j - 1];
  }
  data[i] = x;
  ++currentLength;
}
```

`doubleSpace`

-

seqList::doublespace

```
template<class elemType>
void seqList<elemType>::doublespace()
{
  elemtype *tmp = data;
  maxSize *= 2;
  data = new elemtype[maxSize];
  for (int i = 0 ; i < currentLength ; ++1)
  {
    data[i] = tmp[i];
  }
  delete[] tmp;
}
```

`remove`

seqlist::remove

```
template<class elemType>
void seqList<elemType>::remove(int i)
```

```
{
  if (i < 0 ||i > currentlength)
  {
    throw OutOfBound();
  }
  for (int j = i ; j < currentlength -1 ; ++j)
  {
    data[j] = data[j + 1];
  }
  --currentLength;
}
```

- next next nullptr

-
-
-

-
  -
    - elemType
  -
    -
  -
  - struct

```
template <class elemType>
class sLinkList:public list<elemtype>
{
  private:
    struct node                                    //
    {
      elemType data;
      node *next;
      node(const elemType &x , node *n = nullptr)
      {
        data = x;
        next = n;
      }
      node():next(nullptr){}
      ~node(){}
    };

    node *head;
    int currentLength;
    node *move(int i) const;
  public:
    sLinkList();
    ~sLinkList();
    void clear();
    int length() const
    {
      return currentLength;
    }
    void insert(int i , const elemType &x);
    void remove(int i);
    int search(const elemType &x) const;
    elemType visit(int i) const;
    void traverse() const;
};
```

sLinklist

```
template <class elemType>
sLinkList<elemType>::sLinkList()
{
  head = new node();
  currentLength = 0;
}
```

clear

-
-

sLinkList::clear

```
template <class elemType>
void sLinkList<elemType>::clear()
{
  node *p = head->next , *q;
  while (p != nullptr)        //
  {
    q = p->next;
    delete p;
    p = q;
  }
  currentLength = 0;
}
```

move

-

```
template <class elemType>
struct sLinkList<elemType>::node *sLinkList<elemType>::move(int i) const
{
  node *p = head;
  for (int j = 0 ; j < i ; ++j)
  {
    p = p->next;
  }
  return p;
}
```

insert

1. p i
2. s
3. s next p next
4. p next s

sLinkList::insert

```
template <class elemType>
void sinkList<elemtype>::insert(int i , const elemType &x)
{
  if (i < 0 || i > currentLength)
  {
    throw OutOfBound();
  }
  node *p = move(i-1);
  node *s = new node(x , p->next);
  p->next = s;
  ++currentLength;
}
```

remove

1. pos
2. delp
3. pos next delp next
4. delp

sLinkList::remove

```
template <class elemType>
void sLinkList<elemType>::remove(int i)
{
  if (i < 0 || i >= currentLength)
  {
    throw OutOfBound();
  }
  node *pos = move(i-1);
  node *delp = pos->next;
  pos->next = delp->next;
  delete delp;
  --currentLength;
}
```

search

- x

```
template <class elemType>
int sLinkList<elemType>::search(const elemtype 7x) const
{
  node *p = head -> next;
  int i = 0;
  while (p != nullptr && p->data != x) //
  {
    p = p->next;
    ++i;
  }
  if (p == nullptr)
  {
    return -1;
  }
  else
  {
    return i;
  }
}
```

- p->data != x && p != nullptr p p->data

visit

1. i
2. p->data

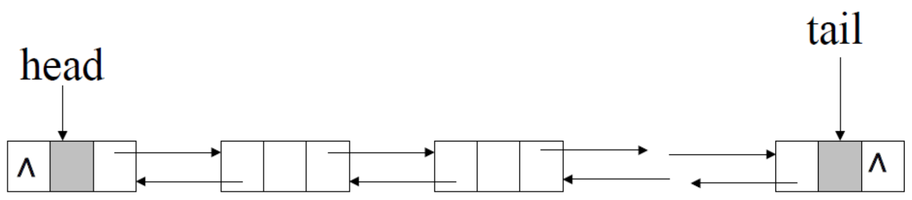
```
template <class elemType>
elemType sLinkList<elemType>::visit(int i) const
{
  if (i < 0 || i >= currentLength)
  {
    throw OutOfBound();
  }
  return move(i)->data;
}
```

traverse

```
template <class elemType>
void sLinkList<elemType>::traverse() const
{
  node *p = head->next;
  while (p != nullptr)
  {
    cout << p->data << ' ';
    p = p->next;
  }
}
```

-
  - next
  - prev

-
  - prev nullptr
  - next
- tail
  - prev
  - next nullptr



```
template <class elemType>
class dLinkList:public list<elemType>
{
  private:
    struct node                                                        //
    {
      elemType data;
      node *next;
      node *prev;
      node(const elemType &x , node *p = nullptr , node *n = nullptr)
      {
        data = x;
        prev = p;
        next = n;
      }
      node():next(nullptr),prev(nullptr){}
      ~node(){}
    };
    node *head , *tail;                                                 //
    int currentLength;                                                  //
    node *move(int i) const;                                            //    i
  public:
    dLinkList();
    ~dLinkList();
    void clear();
    int length() const
    {
      return currentLength;
    }
    void insert(int i , const elemType &x);
    void remove(int i);
    int search(const elemType &x) const;
    elemType visit(int i) const;
    void traverse() const;
};
```

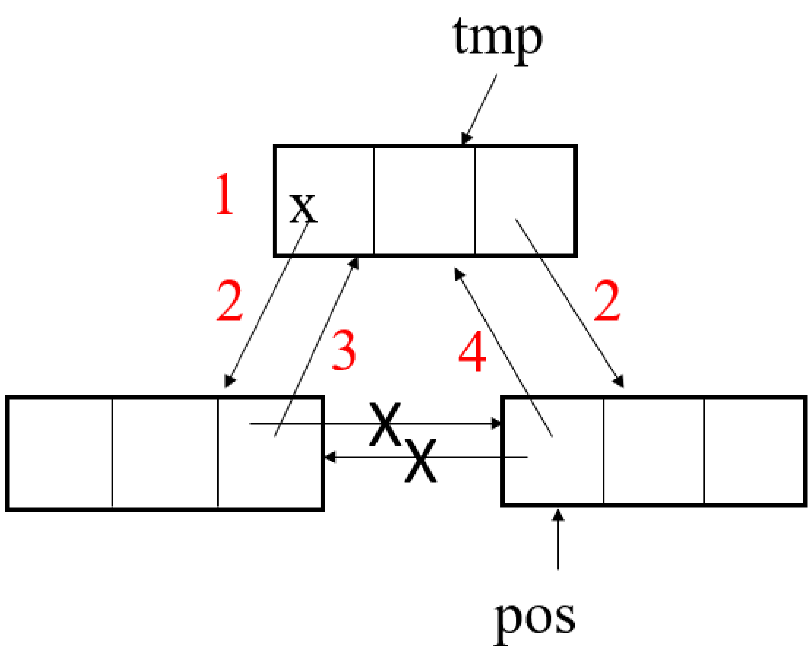
•

dLinkList

```
template <class elemType>
dLinkList<elemType>::dLinkList()
{
  head = new node();
  tail = new node();
  head->next = tail;
  tail->prev = head;
  currentLength = 0;
}
```

insert

1. `tmp` `pos`
2. `tmp` `prev` `pos` `prev` `tmp` `next` `pos`
3. `pos` `next` `tmp`
4. `pos` `prev` `tmp`



```
template <class elemType>
void dLinkList<elemtype>::insert(int i ,const elemTypr &x)
[
  node *pos , *tmp;

  pos = move(i);
  temp = new node(x,pos -> prev , pos);
  pos->prev->next = tmp;
  pos->prev = tmp;

  ++currentLength;
]
```

remove

1. pos
2. pos next pos
3. pos prev pos

dLinkList::remove

```
template <class elemType>
void dLinkList<elemType>::remove(int i)
{
  node *pos = move(i);

  pos->prev->next = pos->next;
  pos->next->prev = pos->prev;

  delete pos;
  --currentLength;
}
```

-

- prev next
-

-

-
- LIFO Last In First Out

- **create()**
- **push(x)** x
- **pop()**
- **top()**
- **isEmpty()** true false

```
template <class elemType>
class stack
{
  public:
    virtual void push(const elemType &x) = 0;       //
    virtual elemType pop() = 0;                     //
    virtual elemType top() const = 0;               //
    virtual bool isEmpty() const = 0;               //
    virtual ~stack() {}                             //
};
```

-
-

```
template <class elemType>
class seqStack:public stack<elemType>
{
  private:
    elemType *data;                     //
    int top_p;                          //
    int maxSize;                        //
    void doubleSpace();
  public:
    seqStack(int initSize = 10);        //
    ~seqStack();                        //
    bool isEmpty() const;               //
    void push(const elemType &x);       //
    elemType pop();                     //
    elemType top() const;                //
};
```

- elem maxSize top_p

```
template <class elemType>
seqStack<elemType>::seqStack(int initSize)
{
  elem = new elemType[initSize];
  maxSize = initSize;
  top_p = -1;
}
```

push

- doubleSpace
- top_p 1
- x

```
template <class elemType>
void seqStack<elemType>::push(const elemType &x)
{
  if (top_p == maxSize - 1)
  {
    doubleSpace();
  }
  elem[++top_p] = x;
}
```

doubleSpace

```
template <class elemType>
void seqStack<elemType>::doubleSpace()
{
  elemType *tmp = elem;
  elem = new elemType[maxSize * 2];
  for (int i = 0 ; i < maxSize ; ++i)
  {
    elem[i] = tmp[i];
  }
  maxSize *= 2;
  delete[] tmp;
}
```

pop

- top_p
- top_p 1

```
template <class elemType>
elemType seqStack<elemType>::pop()
{
  return elem[top_p--];
}
```

top

- top_p

```
template <class elemType>
elemType seqStack<elemType>::top() const
{
  return elem[top_p];
}
```

isEmpty

- top_p

```
template <class elemType>
bool seqStack<elemType>::isEmpty() const
{
  return top_p == -1;
}
```

```
template <class elemType>
seqStack<elemType>::~seqStack()
{
  delete[] elem;
}
```

-
- doublespace

-
-

```
template <class elemType>
class LinkedStack:public stack<elemType>
{
  private:
    struct node
    {
      elemType data;
      node *next;
      node(const elemType &x , node *n = nullptr)
      {
        data = x;
        next = n;
      }
      node():next(nullptr){}
      ~node(){}
    };
    node *top_p;                      //
  public:
    LinkedStack();                   //
    ~LinkedStack();                  //
    bool isEmpty() const;            //
    void push(const elemType &x);    //
    elemType pop();                  //
    elemType top() const;            //
};
```

- top_p   nullptr

```
template <class elemType>
LinkedStack<elemType>::LinkedStack()
{
  top_p = nullptr;
}
```

```
template <class elemType>
LinkStack<elemType>::~LinkStack()
{
  node *tmp;
  while (top_p != nullptr)
  {
    tmp = top_p;
    top_p = top_p->next;
    delete tmp;
  }
}
```

push

-

```
template <class elemType>
void LinkedStack<elemType>::push(const elemType &x)
{
  top_p = new node(x , top_p);
}
```

pop

-

```
template <class elemType>
elemType LinkedStack<elemType>::pop()
{
  node *tmp = top_p;
  elemType x = top_p->data;
  top_p = top_p->next;
  delete tmp;
  return x;
}
```

top

- top_p   data

```
template <class elemType>
elemType LinkedStack<elemType>::top() const
{
  return top_p->data;
}
```

isEmpty

- top_p   nullptr

```
template <class elemType>
bool LinkedStack<elemType>::isEmpty() const
{
  return top_p == nullptr;
}
```

-

-

```
void main()
{
  ...
  r1:f1();
  r2:
  ..
}

void f1()
{
  ...
  t1:f2();
  t2:
  ...
}

void f2()
{
  ...
  ...
}
```

-
-

hanoi

```
void Move(int Height , int FromNeedle ,int ToNeedle ,int UsingNeedle) // FromNeedle   ToNeed
{
  if (Height > 0)
  {
    Move(Height - 1 FromNeedle , UsingNeeedle,ToNeedle); //   n-1  FromNeedle  UsingNeedle
    cout << FromNeedle << "->" << ToNeedle << endl; //      FromNeedle  ToNeedle>
    Move(Height - 1,UsingNeedle , ToNeedle , FromNeedle); //     UsingNeedle  ToNeedle
  }
}
```

Hanio

1.
2.
3.

1.
2.

Fibonacci

```
void printNum(int num)
{
  if (num >= 10)
  {
    printNum(num / 10);
    cout.put(num % 10 + '0');
  }
  else
  {
    cout.put(num + '0');
  }
}
```

1. push(1234)
2. pop(1234) push(4) push(123)
3. pop(123) push(3) push(12)
4. pop(12) push(2) push(1)
5. pop(1) pop(2) pop(3) pop(4)

```
void printNum(int num)
{
  LinkStack<int> s;
  int tmp;
  s.push(num);
  while (!isEmpty())
  {
    tmp = s.pop();
    if (tmp > 9)
    {
      s.push(tmp % 10);
      s.push(tmp / 10);
    }
    else
    {
      cout.put(tmp + '0');
    }
  }
}
```

-
-

1.
2.
3.

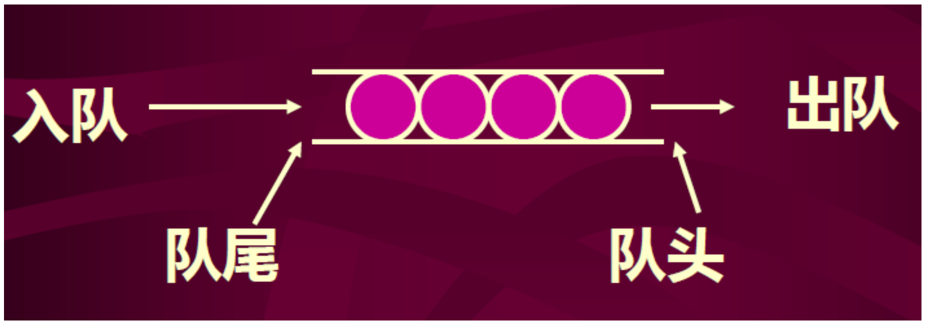
- +ab

- a+b
- ab+

-
-
-

-
-
-
-

-
- FIFO First In First Out
-



- `create()`
- `enQueue(x)` x
- `deQueue()`
- `getHead()`
- `isEmpty()` `true` `false`

```
template <class elemType>
class queue
{
  public:
    virtual void enQueue(const elemType &x) = 0;       //
    virtual elemType deQueue() = 0;                    //
    virtual elemType getHead() const = 0;              //
    virtual bool isEmpty() const = 0;                  //
    virtual ~queue() {}                                 //
};
```

-
- `maxSize - 1`
-
  1.
  2.
  3.

-
  - `rear = (rear + 1) % maxSize`; `elem[rear] = x`
-
  - `front = (front + 1) % maxSize`; `return elem[front]`

    `front`

- `front == rear`
- `(rear + 1) % maxSize == front`

```
template <class elemType>
class seqQueue:public queue<elemType>
{
  private:
    elemType *elem;
    int maxSize;
    int front , rear;
    void doubleSpace();
  public:
    seqQueue(int initSize = 10);        //
    ~seqQueue();                        //
    bool isEmpty();                     //
    void enQueue(const elemType &x);    //
    elemType deQueue();                 //
    elemType getHead();                 //
};
```

- front rear

```
template <class elemType>
seqQueue<elemType>::seqQueue(int initSize)
{
  elem = new elemType[initSize];
  maxSize = initSize;
  front = rear = 0;
}
```

```
template <class elemType>
seqQueue<elemType>::~seqQueue()
{
  delete[] elem;
}
```

enQueue

- doubleSpace
- rear 1

```
template <class elemType>
void seqQueue<elemType>::enQueue(const elemType &x)
{
  if ((rear + 1) % maxSize == front)
  {
    doubleSpace();
  }
  rear = (rear + 1) % maxSize;
  elem[rear] = x;
}
```

doubleSpace

```
template <class elemType>
void seqQueue<elemType>::doubleSpace()
{
  elemType *tmp = elem;
  elem = new elemType[maxSize * 2];
  for (int i = 0 ; i < maxSize ; ++i)
  {
    elem[i] = tmp[(front + i) % maxSize];
  }
  front = 0;
  rear = maxSize - 1;
  maxSize *= 2;
  delete[] tmp;
}
```

deQueue

- front 1
- elem[front]

```
template <class elemType>
elemType seqQueue<elemType>::deQueue()
{
  front = (front + 1) % maxSize;
  return elem[front];
}
```

getHead

- elem[(front + 1) % maxSize]

```
template <class elemType>
elemType seqQueue<elemType>::getHead()
{
  return elem[(front + 1) % maxSize];
}
```

isEmpty

- front rear

```
template <class elemType>
bool seqQueue<elemType>::isEmpty()
{
  return front == rear;
}
```

-
-
- nullptr
- front rear
  - front
  - rear

```
template <class elemType>
class linkQueue:public queue<elemType>
{
  private:
    struct node
    {
      elemType data;
      node *next;
      node(const elemType &x , node *n = nullptr)
      {
        data = x;
        next = n;
      }
      node():next(nullptr){}
      ~node(){}
    };
    node *front , *rear;                //
  public:
    linkQueue();                       //
    ~linkQueue();                      //
    bool isEmpty() ;              //
    void enQueue(const elemType &x);   //
    elemType deQueue();                //
    elemType getHead() const;          //
};
```

- front rear nullptr

```
template <class elemType>
linkQueue<elemType>::linkQueue()
{
  front = rear = nullptr;
}
```

enQueue

1. x
2. rear next x
3. rear x

front rear x

```
template <class elemType>
void linkQueue<elemType>::enQueue(const elemType &x)
{
  if (rear == nullptr)
  {
    front = rear = new node(x);
  }
  else
  {
    rear = rear->next = new node(x);
  }
}
```

deQueue

1. front data
2. front
3.

front rear nullptr

```
template <class elemType>
void LinkQueue<elemType>::deQueue()
{
  node *tmp = front;
  if (front)
  {
    emelType value = front->data;
    front = front->next;
    if (front == nullptr)
    {
      rear = nullptr;
    }
    delete tmp;
    return value;
  }
}
```

getHead

- front data

```
template <class elemType>
elemType linkQueue<elemType>::getHead() const
{
  return front->data;
}
```

isEmpty

- front rear nullptr

```
template <class elemType>
bool linkQueue<elemType>::isEmpty()
{
  return front == nullptr;
}
```

```
template <class elemType>
linkQueue<elemType>::~linkQueue()
{
  node *tmp;
  while (front != nullptr)
  {
    tmp = front;
    front = front->next;
    delete tmp;
  }
}
```

-
  -
  -

-
  -
  -

  -
-
  -
  -
-
-
-
-
-
-
-
-
-
-

- **create()**
- **clear()**
- **IsEmpty()**
- **root()**
- **parent()**
- **child()**
- **remove()**
- **traverse()**

```
template <class T>
class tree
{
  public:
    virtual void clear() = 0;
    virtual bool isEmpty() const = 0;
    virtual T root(T flag) const = 0;
    virtual T parent(T x , T flag) const = 0;
    virtual T child(T x , int i , T flag) const = 0;
    virtual void remove(T x) = 0;
    virtual void traverse() const = 0;
};
```

- Binary Tree

1.
2.
3.
4.
5.

-
-
  -
  -

-
-
-
  -
  -

1.
2.
3.
4.
5.
   (a)
   (b)
   (c)
   (d)

- `create()`
- `clear()`
- `isEmpty()`
- `root()`
- `parent()`
- `lchild()`
- `rchild()`
- `delLeft()`
- `delRight()`
- `traverse()`

•
•

•
•

•
•

•

•
•
•

```
template <class T>
class binaryTree
{
  public:
    virtual void clear() = 0;                     //
    virtual bool isEmpty() const = 0;             //
    virtual T root(T flag) const = 0;             //
    virtual T parent(T x , T flag) const = 0;     //
    virtual T lchild(T x , T flag) const = 0;     //
    virtual T rchild(T x , T flag) const = 0;     //
    virtual void delLeft(T x) = 0;                //
    virtual void delRight(T x) = 0;               //
    virtual void preOrder() const = 0;            //
    virtual void midOrder() const = 0;            //
    virtual void postOrder() const = 0;           //
    virtual void levelOrder() const = 0;          //
};
```

-
-
-

-
  - left data right

- data left parent right

```
template <class T>
struct Node
{
  public:
    Node *left *right;                                                            //
    T data;                                                                       //
    Node():left(nullptr),right(nullptr){}                                         //
    Node(T item , Node *L = nullptr,Node *R = nullptr):data(item),left(L),right(R){}  //
    ~Node(){}                                                                     //
};
```

```
template <class T>
class binaryTree:public tree<T>
{
    friend void printTree(const binaryTree &t, t flag);
  private:
    struct Node
    {
      public:
        Node *left *right;
        T data;
        Node():left(nullptr),right(nullptr){}
        Node(T item , Node *L = nullptr,Node *R = nullptr):data(item),left(L),right(R){}
        ~Node(){}
    }
    Node *root;
  public:
    binaryTree():root(nullptr){}                        //   ,
    binaryTree(T x):root(new Node(x)){}                 //   ,
    ~binaryTree(){}                                     //
    void clear();                                       //
    bool isEmpty() const;                               //
    T Root(T flag) const;                               //
    T lchild(const T &x , T flag) const;                //
    T rchild(const T &x , T flag) const;                //
    void delLeft(const T &x);                           //
    void delRight(const T &x);                          //
    void preOrder() const;                              //
    void midOrder() const;                              //
    void postOrder() const;                             //
    void levelOrder() const;                            //
    coid creatTree(T flag);                             //
    T parent(const T &x , T flag) const                 //
    {
      return flag;
    }
  private:
    Node *Find(const T &x, Node *t) const;                //
    void clear(Node *&x);                                 //
    void preOrder(Node *t) const;                         //
    void midOrder(Node *t) const;                         //
    void postOrder(Node *t) const;                        //
};
```

root nullptr

```
template <class T>
binaryTree<T>::binaryTree()
{
  root = nullptr;
}
```

isEmpty

- root nullptr

```
template <class T>
bool binaryTree<T>::isEmpty() const
{
  return root == nullptr;
}
```

Root Root

```
template <class T>
T binaryTree<T>::Root(T flag)const
{
  if (root == nullptr)
  {
    return flag;
  }
  else
  {
    return root->data;
  }
}
```

1.
2.
3.

```
template <class T>
void binaryTree<T>::preOrder(binaryTree<T>::Node *t) const
{
  if (t == nullptr)
  {
    return;
  }
  else
  {
    cout << t -> data << " ";
    preorder(t -> left);
    preorder(t -> right);
  }
}
```

```
template <class T>
void binaryTree<T>::preOrder() const
{
  cout << "\n   "
  preOrder(root);
}
```

1.
2.
3.

```
template <class T>
void binaryTree<T>::midOrder(binaryTree<T>::Node *t) const
{
  if (t == nullptr)
  {
    return;
  }
  else
  {
    midOrder(t -> left);
    cout << t -> data << " ";
    midOrder(t -> right);
  }
}

template <class T>
void binaryTree<T>::midOrder() const
{
  cout << "\n   "
  midOrder(root);
}
```

1.
2.
3.

```
template <class T>
void binaryTree<T>::postOrder(binaryTree<T>::Node *t) const
{
  if (t == nullptr)
  {
    return;
  }
  else
  {
    postOrder(t -> left);
    postOrder(t -> right);
    cout << t -> data << " ";
  }
}

template <class T>
void binaryTree<T>::postOrder() const
{
  cout << "\n   "
  postOrder(root);
}
```

size

```
template <class T>
int binaryTree<t>::size(binarytree<t>::Node *t) const
{
  if (t = nullptr)
  {
    return 0;
  }
  else
  {
    return 1+sizze(t -> left)+size(t -> right);
  }
}

template <class T>
int binaryTree<T>::size() const
{
  return size(root);
}
```

height

```
template <class T>
int binaryTree<T>::size(binaryTree<T>::Node *t) const
{
  if (t == nullptr)
  {
    return 0;
  }
  else
  {
    int lt = height(t -> left);
    int rt = height(t -> right);
    return (lt > rt ? lt : rt) + 1;
  }
}

template <class T>
int binaryTree<T>::height() const
{
  return height(root);
}
```

-

```
template <class T>
void binaryTree<T>::levelOrder() const
{
  linkQueue<Node *> que;
  Node *tmp;
  cout << "\n   ";
  que.enQueue(root);
  while (!que.isEmpty())
  {
    tmp = que.deQueue();
    cout << tmp -> data << " ";
    if (tmp.left)
    {
      que.enQueue(tmp -> left);
    }
    if (tmp.right)
    {
      que.enQueue(tmp -> right);
    }
  }
}
```

clear

-

```
template <class T>
void binaryTree<T>::clear(Node *&t)
{
  if (t == nullptr)
  {
    return;
  }
  else
  {
    clear(t -> left);
    clear(t -> right);
    delete t;
    t = nullptr;
  }
}
```

```
template <class T>
void binaryTree<T>::clear()
{
  clear(root);
}
```

- clear

```
template <class T>
binaryTree<T>::~binaryTree()
{
  clear();
}
```

Find

- x
- x
- Find x
- Find x

```
template <class T>
struct binaryTree<T>:: Node *binaryTree<T>::Find(const T &x ,binaryTree<T>::Node *t) const
{
  if (t == nullptr)
  {
    return nullptr;
  }
  else if (t -> data == x)
  {
    return t;
  }
  else
  {
    if (tmp = Find(x,t->left))
    {
      return tmp;
    }
    else
    {
      return Find(x,t->right);
    }
  }
}
```

delLeft

```
template <class T>
void binaryTree<T>::delLeft(const T &x)
```

```
{
  Node *tmp = Find(x,root);
  if (tmp == nullptr)
  {
    return;
  }
  clear(tmp -> left);
}
```

delRight

```
template <class T>
void binaryTree<T>::delRight(const T &x)
{
  Node *tmp = Find(x,root);
  if (tmp == nullptr)
  {
    return;
  }
  clear(tmp -> right);
}
```

lchild

```
template <class T>
T binaryTree<T>::lchild(const T &x ,T flag) const
{
  Node *tmp = Find(x,root)
  if (tmp == nullptr || tmp -> left ==nullptr)
  {
    return flag;
  }
  else
  {
    return tmp -> left;
  }
}
```

rchild

```
template <class T>
T binaryTree<T>::rchild(const T &x ,T flag) const
{
  Node *tmp = Find(x,root)
  if (tmp == nullptr || tmp -> right ==nullptr)
  {
    return flag;
  }
  else
  {
    return tmp -> right;
  }
}
```

createTree

-
  1.
  2. flag
-
  -
  -

```
template <class T>
void binaryTree<T>::createTree(T flag)
{
  linkQueue<Node *> que;
  Node *tmp;
  T x , ldata , rdata;

  //   flag
  cout << "\n   "
  cin >> x;
  root = new Node(x);
  que.enQueue(root);
  while (!que.isEmpty())
  {
    tmp = que,deQueue();
    cout << "\n " << tmp -> data << "   "<<flag<<"    ";
    cin >> ldata >> rdata;
    if (ldata != flag)
    {
      tmp -> left = new Node(ldata);
      que.enQueue(tmp -> left);
    }
    if (rdata != flag)
    {
      tmp -> right = new Node(rdata);
      que.enQueue(tmp -> right);
    }
  }
  cout << "that's good!\n";
}
```

printTree

-

```
template <class T>
void printTree(const binaryTree<T> &t,T flag)
{
  linkQueue<T> que;
  que.enQueue(t.root());
  while(!que.isEmpty())
  {
    T tmp = que.deQueue();
    T l = tmp.lchild(tmp , flag);
    T r = tmp.rchild(tmp , flag);
    cout << p << " " << l << " " << r << endl;
    if (l != flag)
    {
      que.enQueue(l);
    }
    if (r != flag)
    {
      que.enQueue(r);
    }
  }
}
```

1.
2.
3.
4.

```
template <class T>
void binaryTree<T>::preOrder(const binaryTree<T>::Node *t) const
{
  linkStack<Node *> s;
  Node *tmp = t;

  cout << "\n   ";
  while (!s.isEmpty())
  {
    tmp = s.pop();
    cout << tm -> data << " ";
    if (tmp -> right != nullptr)
    {
      s.push(tmp -> right);
    }
    if (tmp -> left != nullptr)
    {
      s.push(tmp -> left);
    }
  }
}
```

1.
2.
3.
4.
5.
6.

-
    -
    -
-
    -
    -
-

StNode

```
struct StNode
{
  Node *node;
  int TimesPop;
  StNode(Node *N=nullptr):node(N),TimesPop(0){}
};
```

```
template <class T>
void binaryTree<T>::midOrder(const binaryTree<T>::Node *t) const
{
  linkStack<Node *t> s;
  StNode current(root);

  cout << "\n   ";
  s.push(current);
  while(!s.isEmpty())
  {
    current = s.pop();
    if (++current.TimesPop == 2)
    {
      cout << current.node -> data;
      if (current.node -> right != nullptr)
      {
        s.push(StNode(current.node -> right));
      }
    }
    else
    {
      s.push(current);
      if (current.node -> left != nullptr)
      {
        s.push(StNode(current.node -> left));
      }
    }
  }
}
```

1.
2.
3.
4.
5.

```
template <class T>
void binaryTree<T>::postOrder(const binaryTree<T>::Node *t) const
{
  linkStack<Node *> s;
  StNode current(root);

  cout << "\n   ";
  s.push(current);
  while(!s.isEmpty())
  {
    current = s.pop();
    if (++current.TimesPop == 3)
    {
      cout << current.node -> data;
      continue;
    }
    else
    {
      s.push(current);
      if (current.node -> right != nullptr)
      {
        s.push(StNode(current.node -> right));
      }
      if (current.node -> left != nullptr)
      {
        s.push(StNode(current.node -> left));
      }
    }
  }
}
```

-
-
    -
    -
    -
    -

-
-
-
-

huffmanTree

1.
2.
3.
    (a)
    (b)
4.

-
- `0` `1`

1. ,
2.
   (a)
   (b) getCode

-
-
-
-

```
template <class Type>
class hfTree
{
  private:
    struct Node
    {
      Type data; //
      int weight; //
      int parent; //
      int left , right; //
    };

    Node *elem;
    int length;

  public:
    struct hfCode
    {
      Type data; //
      string code; //
    };

    hfTree(const Type *v , const int *w , int size);
    void getCode(hfCode result[]);
    ~hfTree()
    {
      delete [] elem;
    }
};
```

```
template <class Type>
hfTree<Type>::hfTree(const Type *v , const int *w , int size)
{
  const int MAX_INT = 32767;
  int min1 , min2; //
  int x , y ; //

  /*    */
  length = 2 * size ;                                             //       $2\text{size}-1$
  elem = new Node[length];                                        //      $2\text{size}$
  for (int i = size ; i < length ; i++)                           //    `elem`        $\text{
  {
    elem[i].weight = w[i-size]; //
    elem[i].data = v[i-size]; //
    elem[i].parent = elem[i].left = elem[i].right = 0;            //             `0`
  }

  /*      */
  for (int i = size - 1 ; i > 0 ; i--)                            //    `size-1`
  {
    min1 = min2 = MAX_INT;                                        //
    x = y = 0;
    for (int j = i + 1 ; j < length ; j++)                        //
    {
      if (elem[j].parent == 0)                                    //
      {
        if (elem[j].weight < min1)                                //
        {
          min2 = min1;
          min1 = elem[j].weight;
          x = y;
          y = j;
        }
        else if(elem[j].weight < min2)
        {
          min2 = elem[j].weight;
          x = j;
        }
      }
    }
    elem[i].weight = min1 + min2;                                 //
    elem[i].left = x;                                             //
    elem[i].right = y;                                            //
    elem[x].parent = i;                                           //
    elem[y].parent = i;                                           //
  }
}
```

getCode

```
template <class Type>
void hdTree<Type>::getcode(hfCode result[])
{
  int size = length / 2 ;
  int p , s ;
  for (int i = size; i < length; ++i)
  {
    result[i -size].data = elem[i].data;
    result[i -size].code = "";
    p = elem[i].parent; s = i;
    while (p)
    {
      if (elem[p].left == s)
      {
        result[i -size].code = '0' + result[i -size].code;
      }
      else
      {
        result[i -size].code = '1' + result[i -size].code;
      }
      s = p;
      p = elem[p].parent;
    }
  }
}
```

-
-

-
-
  -

–

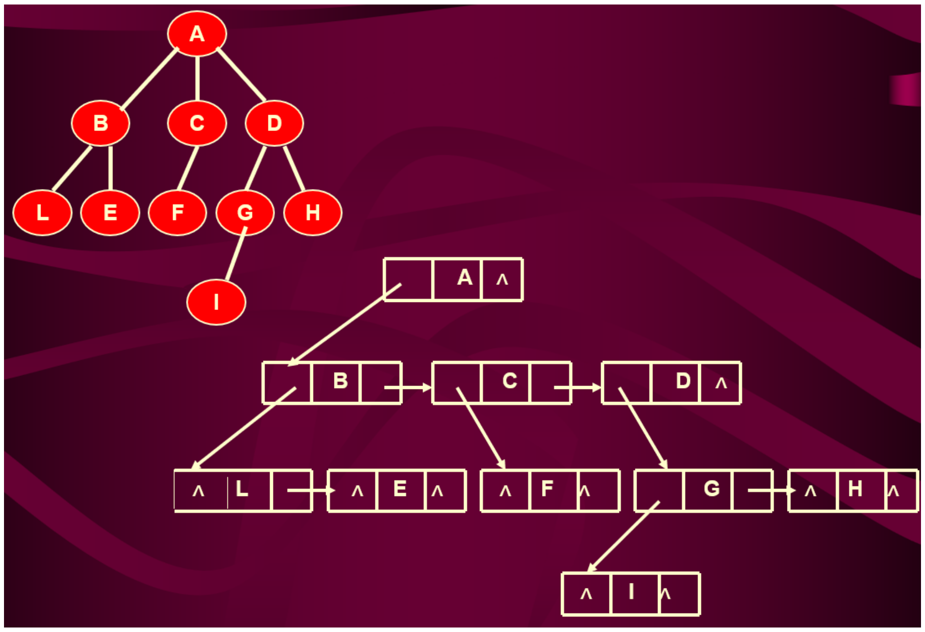
•

•

•

•

•



•

–

–

•

1.

2.

1.
2.

1.
2.

•
•

•
•
  –
  –

1.
2.

•
•
•

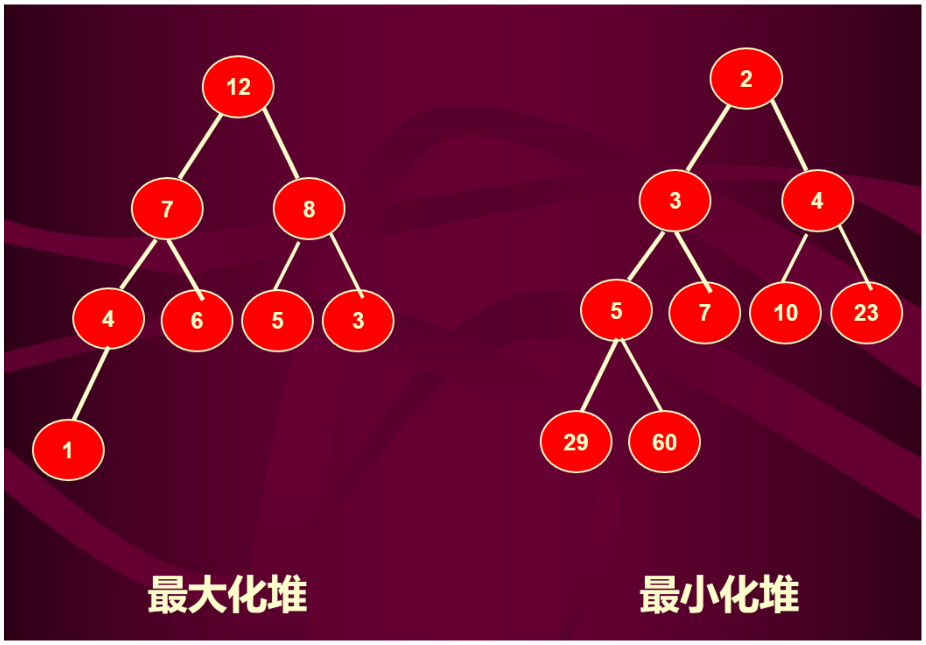
1.

2.

•
•
  1.

2.

-



-
  -
-
  -
  -

-
- 1
  - 1
  - 1
  -

```
template <class T>
class priorityQueue:public quque<T>
{
  private:
    int currentSize;
    T *array;
    int maxSize;
    void doubleSpace();
    voif buildHeap();  //    priorityQueue()
    void percolateDown(int hole); //
  public:
    priorityQueue(int capacity = 100) //
    {
      array = new T[capacity];
      maxSize = capacity;
      currentSize = 0;
    }
    priorityQueue(const T data[] , int size); //
    ~priorityQueue(); //
    bool isEmpty() const //
    {
      return currentSize == 0;
    }
    void enQueue(const T &x); //
    T deQueue(); //
    T getHead() const //
    {
      return array[1];
    }
};
```

enQueue

-
-
-

```
template <class T>
void priorityQueue<T>::enQueue(const T &x)
{
  if (currentSize == maxSize - 1)
  {
    doubleSpace();
  }

  //
  int hole = ++currentSize;
  for (;hole > 1 && x < array[hole/2] ; hole /= 2)
  {
    array[hole] = array[hole/2];
  }
  array[hole] = x;
}
```

-
-

deQueue

-
-

```
template <class T>
T priorityQueue<T>::deQueue()
{
  T minItem;
  minItem = array[1];
  array[1] = array[currentSize--];
  percolateDown(1);
  return minItem;
}
```

percolateDown

```
template <class T>
void priorityQueue<T>::precolateDown(int hole)
{
  int child;
  T tmp = array[hole];

  for (;hole * 2 <= currentSize; hole = child)
  {
    child = hole * 2;
    if (child != currentSIze && array[child + 1] < array[child])
    {
      child++;
    }
    if (array[child] < tmp)
    {
      array[hole] = array[child];
    }
    else
    {
      break;
    }
  }
  array[hole] = tmp;
}
```

buildHeap

-

- buildHeaap
-

percolateDown

1.
2.

```
template<calss KEY , class OTHER>
struct SET
{
  KEY key; //
  OTHER other; //
}
```

-
-

key

-
-
-

-
-
    -
    -
    -
    -

-
-
-
- seqList C++

-
-
-
-

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY, OTHER> data[] , int size , const KEY &x)
{
  data[0].key = x;
  for (int i = size ; x != data[i].key; --i)
  {
    return i;
  }
}
13
```

-
-
-

```
template <class KEY, class OTHER>
int seqSearch(SET<KEY , OTHER> data[] , int size , const KEY &x)
{
  data[0].key = x;
  for (int i = size ; x < data[i].key; --i);
  if (x == data[i].key)
  {
    return i;
  }
else
{
  return 0;
}
```

1.
2.
3.
4.

-

```
template <class KEY, class OTHER>
int binarySearch(SET<KEY , OTHER>data[] , int size , const KEY &x)
{
  int low = 1,  high = size, mid;
    while (low <= high)
    {                //
      mid = (low + high) / 2;          //
      if ( x == data[mid].key )
      {
        return mid;
      }
      if (x < data[mid].key)
      {
        high = mid - 1;
      }
      else low = mid + 1;
    }
  return 0;
}
```

-
-
-

-
-
    1.
    2.

```
template <class KEY, class OTHER>
class dynamicSearchTable
{
  public:
    virtual SET<KEY, OTHER> *find(const KEY &x) const = 0;
    virtual void insert(const SET<KEY, OTHER> &x) = 0;
    virtual void remove(const KEY &x) = 0;
    virtual ~dynamicSearchTable() {};
};
```

- p
  - p p
  - p p
  - p
-
-
-

```
template <class KEY, class OTHER>
class BinarySearchTree:public dynamicSearchTable<KEY , OTHER>
{
  private:
    struct BinaryNode
    {
      SET<KEY, OTHER> data;
      BinaryNode*left;
      BinaryNode*right;
      BinaryNode( const SET<KEY, OTHER> & thedata,
      BinaryNode *lt = nullptr , BinaryNode *rt = nullptr):data(thedata) , left(lt) , right
    };
    BinaryNode*root;

  public:
    BinarySearchTree();
    ~BinarySearchTree();
    SET<KEY, OTHER> *find(const KEY &x) const ;
    void insert(const SET<KEY , OTHER> &x );
    void remove(const KEY &x);

  private:
    void insert(const SET<KEY , OTHER> &x, BinaryNode *&t);
    void remove(const KEY &x , BinaryNode *&t);
    SET<KEY , OTHER> *find(const KEY &x , BinaryNode *t ) const;
    void makeEmpty(BinaryNode *t);//     clear
};
```

1.
2.
3.
4.

- find(const KEY &x)

```
template <class KEY, class OTHER>
SET<KEY, OTHER> *BinarySearchTree<KEY, OTHER>::find(const KEY &x ) const
{
  return find( x, root );
}
```

- find(const KEY &x, BinaryNode *t)

```
template <class KEY, class OTHER>
SET<KEY, OTHER> *BinarySearchTree<KEY, OTHER>::find(const KEY &x, BinaryNode *t ) const
{
  if (t == nullptr || t->data.key == x)
  {
    return (SET<KEY , OTHER> *)t;//
  }
  if(x < t->data.key)
  {
    return find(x , t->left);
  }
  else
  {
    return find(x , t->right );
  }
}
```

-

-
    -

- insert(const SET<KEY, OTHER> &x)

```
template <class KEY, class OTHER>
void BinarySearchTree<KEY, OTHER>::insert(const SET<KEY , OTHER> &x)
{
  insert(x, root);
}
```

- insert(const SET<KEY, OTHER> &x, BinaryNode *&t)

```
template <class KEY, class OTHER>
void BinarySearchTree<KEY , OTHER>::insert(const SET<KEY , OTHER> &x, BinaryNode *&t)
{
  if(t == nullptr)
  t = new BinaryNode(x , nullptr , nullptr);
  else if(x.key< t->data.key)
  {
    insert(x, t->left);
  }
  else if(x.key > t->data.key)
  {
    insert(x , t->right);
  }
  else
  {
    cout << x.key << "is exist" << endl;
  }
}
```

-
    1.
    2.
    3.
        (a)
        (b)
        (c)

- remove(const KEY &x)

```
template <class KEY, class OTHER>
void BinarySearchTree<KEY, OTHER>::remove(const KEY &x)
{
  remove(x , root);
}
```

- remove(const KEY &x, BinaryNode *&t)

```
template <class KEY, class OTHER>
void BinarySearchTree<KEY , OTHER>::remove(const KEY &x , BinaryNode *&t)
{
  if(t == nullptr)
  {
    cout << x.key << "is not exist" << endl;
  }
  else if(x.key < t->data.key)
  {
    remove(x , t->left);
  }
  else if(x.key > t->data.key)
  {
    remove(x , t->right);
  }
  else if(t->left != nullptr && t->right != nullptr)
  {
    BinaryNode *p = t->right;
    while(p->left != nullptr)
    {
      p = p->left;
    }
    t->data = p->data;
    remove(p->data.key , t->right);
  }
  else
  {
    BinaryNode *oldNode = t;
    t = (t->left != nullptr) ? t->left : t->right;
    delete oldNode;
  }
}
```

-
-
-

AVL

AVL

-
  - 1
-
  -
  -
-
  -

-
-
  -
  -

- 1
-
  1. 0
  2. 1
  3.
  4.

-
-
-

AVL

-
-
  -
  -

AVL

```
template <clas KEY,class OTHER>
class AvlTree:public dynamicSearchTable<KEY , OTHER>
{
    struct AvlNode
    {
      SET<KEY , OTHER> data ; //
      AvlNode *left , *right; //
      int height;            //

      AvlNode(const SET<KEY , OTHER> &element , AvlNode *lt ,AvlNode   *rt , int h = 1):data
    };

    AvlNode *root; //
  public:
    AvlTree() //
    {
      root = nullptr;
    }
    ~AvlTree() //
    {
      makeEmpty(root);
    }
    SET<KEY, OTHER> *find(const KEY &x) const;
    void remove(const KEY & x);
  private:
    void insert( const SET<KEY, OTHER> & x, AvlNode * & t ) ; //
    bool remove( const KEY & x, AvlNode * & t ) ;
    void makeEmpty( AvlNode *t );
    int height(AvlNode *t) const  //
    {
      return t == nullptr ? 0 : t -> height;
    }
    void LL( AvlNode * & t ); //
    void RR( AvlNode * & t ); //
    void LR( AvlNode * & t ); //
    void RL( AvlNode * & t ); //
    int max(int a, int b)     //
    {
      return a > b ? a : b;
    }
    bool adjust(AvlNode *&t, int subTree); //
};
```

AVL

-
-
- AVL

1.
2.
   (a) x
   (b) x
   (c) x
3.

```
template <class KEY,class OTHER>
SET<KEY, OTHER> *AvlTree<KEY, OTHER>::find(const KEY & x) const
{
  AvlNode *t = root;
  while (t!=nullptr && t->data.key != x)
  {
    if (x < t->data.key)
    {
      t = t->left;
    }
    else
    {
      t = t->right;
    }
  }
  if (t == nullptr)
  {
    return nullptr;
  }
  else
  {
    return t->data;
  }
}
```

AVL

-
  -
  -

- LL

- LR
- RL
- RR

-
  -
  -

LL

LL

-
  1.
  2.
- RR
-

LR

LR

-
  1.
  2.
- RL
-

insert

```
template <class KEY,class OTHER>
void AvlTree<KEY, OTHER>::insert(const SET<KEY, OTHER> & x, AvlNode * & t)
{
  if (t == nullptr)
  {
    t = new AvlNode(x , nullptr , nullptr);
  }
  else if (x.key < t->data.key)
  {
    insert(x , t->left);
    if (height(t->left) - height(t->right) == 2)
    {
      if (x.key < t->left->data.key)
      {
        LL(t);
      }
      else
      {
        LR(t);
      }
    }
  }
  else if (x.key > t->data.key)
  {
    insert(x , t->right);
    if (height(t->right) - height(t->left) == 2)
    {
      if (x.key > t->right->data.key)
      {
        RR(t);
      }
      else
      {
        RL(t);
      }
    }
  }
  t->height = max(height(t->left) , height(t->right)) + 1; //
}
```

LL

```
template <class KEY,class OTHER>
void AvlTree<KEY,OTHER>::LL(AvlNode *& t)
{
  AvlNode *t1 = t->left;
  t->left = t1->right;
  t1->right = t;
  t->height = max(height(t->left) , height(t->right)) + 1;
  t1->height = max(height(t1->left) , height(t1->right)) + 1;
  t = t1;
}
```

RR

```
template <class KEY,class OTHER>
void AvlTree<KEY,OTHER>::RR(AvlNode *& t)
{
  AvlNode *t1 = t->right;
  t->right = t1->left;
  t1->left = t;
  t->height = max(height(t->left) , height(t->right)) + 1;
  t1->height = max(height(t1->left) , height(t1->right)) + 1;
  t = t1;
}
```

LR

```
template <class KEY,class OTHER>
void AvlTree<KEY,OTHER>::LR(AvlNode *& t)
{
  RR(t->left);
  LL(t);
}
```

RL

```
template <class KEY,class OTHER>
void AvlTree<KEY,OTHER>::RL(AvlNode *& t)
{
  LL(t->right);
  RR(t);
}
```

`insert`

AVL

1. AVL `x`
2.

-
-
- bool `true`

a Sitiationa

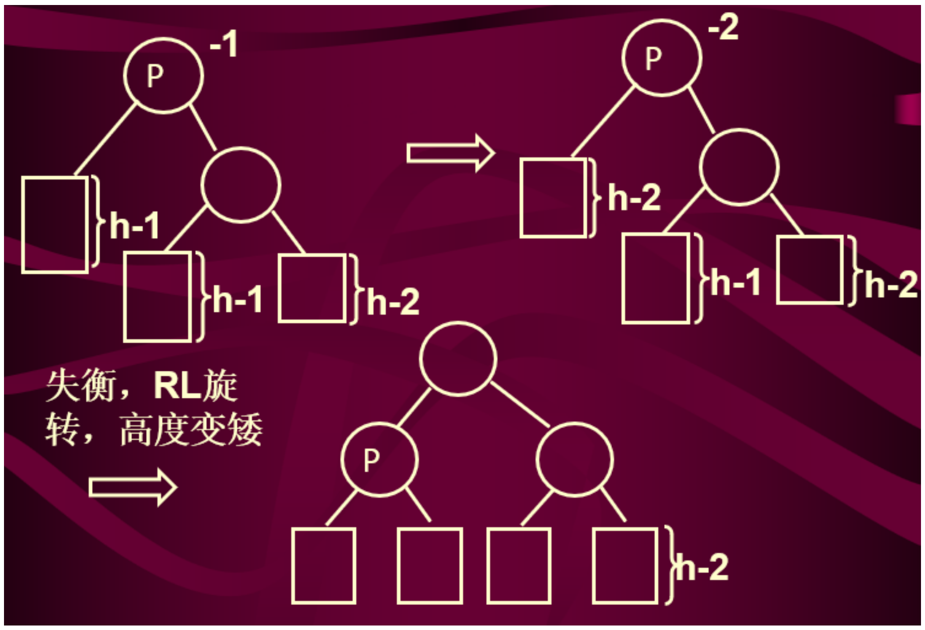
`true`

b Situationb

`false`

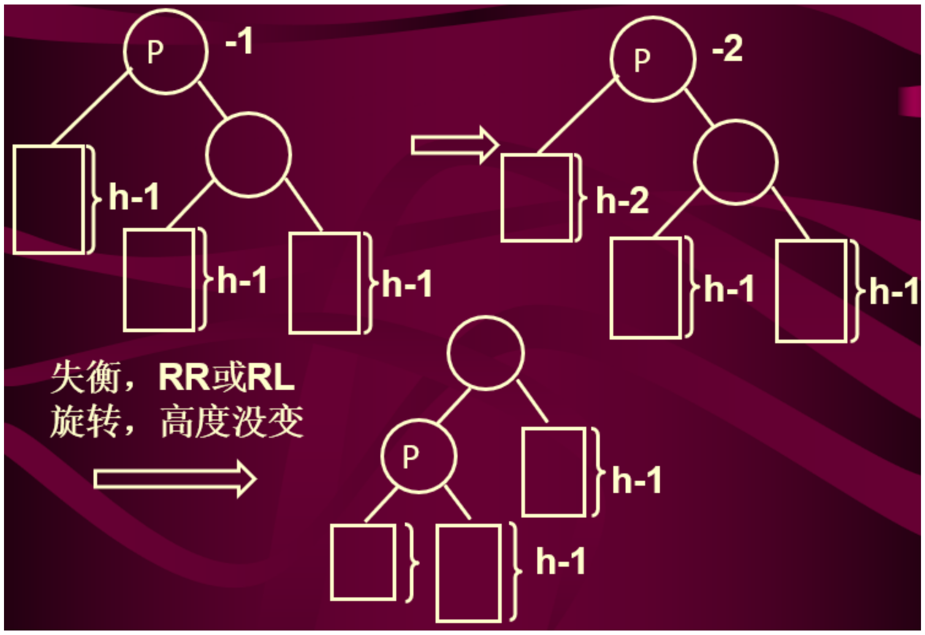
c Situationc

RR `false`



d

RL `false`



e

RR RL `true`

- false
- true true 5

remove

```
template<class KEY,class OTHER>
void AvlTree<KEY , OTHER>::remove(const KEY &x , AvlNOde *&x)
{
  if (t = nullptr) //
  {
    return true;
  }
  if (x == t-> data.key)
  {
    if (t->left == nullptr || t-> right == nullptr)
    {
      AvlNode *oldNode = t;
      t = (t->left == nullptr) ? t->right : t->left; //
      delete oldNode;
      return false; //       `false`
    }
    else
    {
      AvlNode *tmp = t-> right;
      while (tmp->left != nullptr) //
      {
        tmp = tmp->left;
      }
      t->data = tmp.data;
      if (remove(tmp->data,key , t->right))
      {
        return adjust(t,1);
      }
    }
  }
  if (x < t->data,key)
  {
    if (remove(x,t->left))
    {
      teturn true;
    }
    return adjust(t,0);
  }
  else
  {
    if (remove(x , t->right))
    {
      return true;
    }
    return adjust(t,1);
  }
}
```

adjust

-
-
-
    - true
    - false
-
    - AVlNode *&t
    - int subTree t
        * 0
        * 1

```
template<class KEY , classs OTHER>
bool AvlTree<KEY , OTHER>::adjust(AvlNode *&t , int SubTree)
{
  if (subTree) //
  {
    if (height(t->left) - height(t->right) == 1) // Situation a
    {
      return true;
    }
    if (height(t->right) == height(t->left)) // Situation b
    {
      return false;
    }
    if (height(t->left->right) > height(t->left->left)) // Situation d
    {
      LR(t);
      return false;
    }
    LL(t); // Situation c and e
    if (height(t->left) == height(t->right))s
    {
      return false;
    }
    else
    {
      return true;
    }
  }
  else //
  {
    if (height(t->right) - height(t->left) == 1) // Situation a
    {
      return true;
    }
    if (height(t->right) == height(t->left)) // Situation b
    {
      return false;
    }s
    if (height(t->right->left) > height(t->right->right)) // Situation d
    {
      RL(t);
      return false;
    }
    RR(t); // Situation c and e
    if (height(t->right) == height(t->left))
    {
      return false;
    }
    else
    {
      return true;
    }
  }
}
```

-
- KEY
-

1.

2. `insert(i)`
   `i` `a[i.key]`
3. `find(i)`
   `a[i.key]`
4. `remove(i)`
   `a[i.key]`

hush function

-
  D key H
-
  -
  -

-
-

-
  -
    *
    *
    *
  -

-
  - insert
  - remove
  - find
-
-
  - 0
  - 1
  - 2

```
template <class KEY, class OTHER>
class closeHashTable:public dynamicSearchTable<KEY, OTHER>
{
  private:
    struct node  //
    {
      SET <KEY, OTHER> data;
      int state; //0 -- empty 1 -- active 2 -- deleted
      node()
      {
        state = 0;
      }
    };
    node *array;

    int size;
    int (*key)(const KEY &x);//
    static int defaultKey(const int &x)
    {
      return x;
    }
  public:
    closeHashTable(int length = 101, int (*f)(const KEY &x) = defaultKey)
    ~closeHashTable()
    {
      delete [] array;
    }
    SET<KEY, OTHER> *find(const KEY &x) const;
    void insert(const SET<KEY, OTHER> &x);
    void remove(const KEY &x);
};
```

-

```
template <class KEY, class OTHER>
closeHashTable<KEY, OTHER>::closeHashTable(int length, int (*f)(const KEY &x))
{
  size = length;
  array = new node[size];
  key = f; //   f
}
```

- insert

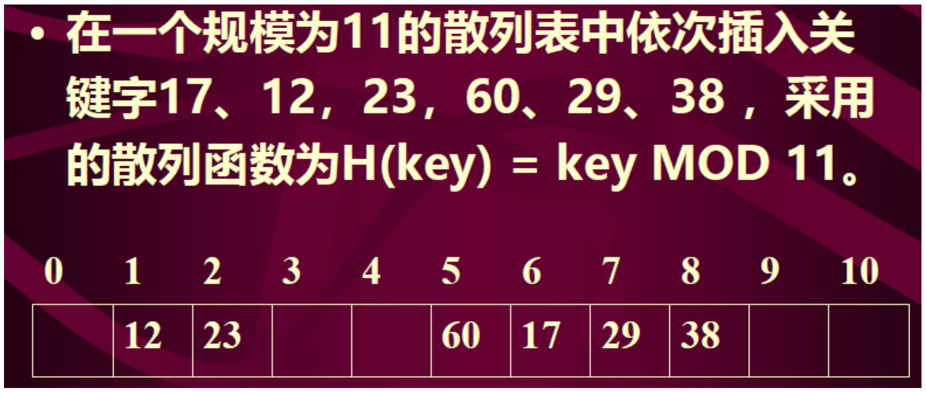
```
template <class KEY, class OTHER>
void closeHashTable<KEY, OTHER>::insert(const SET<KEY, OTHER> &x)
{
  int initPos, pos ;
  initPos= pos = key(x.key) % size; //%size
  do
  {
    if (array[pos].state != 1)
    { // 0 2
      array[pos].data = x;
      array[pos].state = 1;
      return;
    }
    pos = (pos+1) % size;
  } while (pos != initPos);
}
```

- remove

```
template <class KEY, class OTHER>
void closeHashTable<KEY, OTHER>::remove(const KEY &x)
{
  int initPos, pos ;
  initPos= pos = key(x) % size;
  do
  {
    if (array[pos].state == 0) return; //
    if (array[pos].state == 1 && array[pos].data.key== x)//
    {
      array[pos].state = 2;
      return;
    }
    pos = (pos+1) % size; //
  } while (pos != initPos);
}
```

- find

```
template <class KEY, class OTHER>
SET<KEY, OTHER> *closeHashTable<KEY, OTHER>::find(const KEY &x) const
{
  int initPos, pos ;
  initPos = pos = key(x) % size;
  do
  {
    if (array[pos].state == 0) //
    {
      return nullptr;
    }
    if (array[pos].state == 1 && array[pos].data.key == x) //
    {
      return (SET<KEY,OTHER> *)&array[pos];
    }
    pos = (pos+1) % size;
  } while (pos != initPos);
}
```



• 在一个规模为11的散列表中依次插入关键字17、12，23，60、29、38，采用的散列函数为H(key) = key MOD 11。

0	1	2	3	4	5	6	7	8	9	10
	12	23			60	17	29	38		

-

-
-

- M

-
-
-

```
template <class KEY , class OTHER>
class openHashTable:public dynamicSearchTable<KEU , OTHER>
{
  private:
    struct node
    {
      SET<KEY , OTHER> data;
      node *next;
      node (const SET<KEY , OTHER> &d , nde *n = nullptr)
      {
        data = d;
        next = n;
      }
      node()
      {
        next = nullptr;
      }
    };
    node **array; //
     int size;
     static int defaultKry(const int &x)
     {
        return x;
     }
  public:
    openHashTable(int length = 101,int (*f)(const KEY &x) = defaultKey);
    ~openashTable();
    SET<KEY,OTHER> *finf(const KEY &x) const;
    void insert(const SET<KEY,OTHER> &x);
    void remove(const KEY &x);
};
```

-

```
template <class KEY, class OTHER>
openHashTable<KEY, OTHER>::~openHashTable()
{
  node *p, *q;
  for (int i = 0; i< size; ++i)
  {
    p = array[i];
    while (p!=nullptr)
    {
      q= p->next; delete p; p = q;
    }
  }
  delete [] array
}
```

- insert

```
template <class KEY, class OTHER>
void openHashTable<KEY, OTHER>::insert(const SET<KEY, OTHER> &x)
{
  int pos;
  node *p;
  //
  pos = key(x.key) % size;
  array[pos] = new node(x, array[pos]);
}
```

- remove

```
template <class KEY, class OTHER>
void openHashTable<KEY, OTHER>::remove(const KEY &x)
{
  int pos ;
  node *p, *q;
  pos = key(x) % size;
  if (array[pos] == nullptr)
  {
    return;
  }
  p = array[pos];
  if (array[pos]->data.key== x)
  { //
    array[pos] = p->next;
    delete p;
    return;
  }
  while (p->next != nullptr && !(p->next->data.key== x))
  {
    p = p->next;
  }
  if (p->next != nullptr)
  {
    q = p->next;
    p->next = q->next;
    delete q;
  }
}
```

- find

```
template <class KEY, class OTHER>
SET<KEY, OTHER> *openHashTable<KEY, OTHER>::find(const KEY &x) const
{
  int pos ;
  node *p;
  pos = key(x) % size;
  p = array[pos];
  while (p != nullptr && !(p->data.key == x))
  {
    p = p->next;
  }
  if (p == nullptr)
  {
    return nullptr;
  }
  else
  {
    return (SET<KEY, OTHER> *)p;
  }
}
```

•
•   :
•
    –
    –

n-1

```
template <class KEY, class OTHER>
void simpleInsertSort(SET<KEY, OTHER>a[], int size)
{
  int k;
  SET<KEY, OTHER> tmp;
  for (int j=1; j<size; ++j)
  {
  tmp = a[j];
  for ( k = j-1; tmp.key < a[k].key && k >= 0; --k)
  {
    a[k+1] = a[k];
  }
  a[k+1] = tmp;
  }
}
```

•
•
•
    –
    –
    –

-

-
  -
  -
  -
-
-

-
-
-

-
-
- Knuth
-
-
-

```
template <class KEY, class OTHER>
void shellSort(SET<KEY, OTHER> a[], int size)
{
  int step, i, j;
  SET<KEY, OTHER> tmp;
  for (step = size/2; step > 0; step /= 2) //step
  {
    for (i = step; i < size; ++i)
    {
      tmp = a[i];
      for (j = i -step; j >= 0 && a[j].key > tmp.key; j -= step)
      {
        a[j+step] = a[j];
      }
      a[j+step] = tmp;
    }
}
```

1.
2.
3.

1.
2.
3.

-
-
-

```
template <class KEY, class OTHER>
void simpleSelectSort(SET<KEY, OTHER> a[], int size)
{
  int i, j, min;
  SET<KEY, OTHER> tmp;
  for (i = 0; i < size -1; ++i)
  {
  min = i;
  for (j = i+1; j < size; ++j)
  {
    if (a[j].key < a[min].key)
    {
      min = j;
    }
  }
  tmp = a[i]; a[i] = a[min]; a[min] = tmp;
  }
}
```

1.
2. deQuqeue

-
-

```
template <class KEY, class OTHER>
void heapSort(SET<KEY, OTHER> a[], int size)
{
  int i;
  SET<KEY, OTHER> tmp; //
  for( i = size / 2 -1; i >= 0; i--)
  {
    percolateDown( a, i, size );
  }
  // n-1 deQueue
  for ( i = size -1; i > 0; --i)\
  {
  tmp = a[0]; a[0] = a[i]; a[i] = tmp; //delete a[0]
  percolateDown( a, 0, i );
  }
}
```

- precolateDown

```
template <class KEY, class OTHER>
void percolateDown( SET<KEY, OTHER> a[], int hole, int size)
{
  int child;
  SET<KEY, OTHER> tmp= a[ hole ];
  for( ; hole * 2 + 1 < size; hole = child )
  {
    child = hole * 2 + 1;
    if( child != size -1 && a[ child + 1 ].key > a[ child ].key )
    {
      child++;
    }
    if( a[ child ].key >tmp.key)
    {
      a[ hole ] = a[ child ];
    }
    else
    {
      break;
    }
  }
  a[ hole ] = tmp;
}
```

2

```
template <class KEY, class OTHER>
void bubbleSort(SET<KEY, OTHER> a[], int size)
{
  int i, j;
  SET<KEY, OTHER> tmp;
  bool flag = true; //
  for (i = 1; i < size&& flag; ++i)
  { //size-1
    flag = false;
    for (j = 0; j < size-i; ++j) // i
    if (a[j+1].key < a[j].key)
    {
      tmp = a[j]; a[j] = a[j+1]; a[j+1] = tmp;
      flag = true;
    }
  }
}
```

-
-

-
-
-

1. high low K low
2. high K high K
3. k low high low K
4. low high 2 low high K

```
template <class KEY, class OTHER>
int divide( SET<KEY, OTHER> a[], int low, int high)
{
  SET<KEY, OTHER> k = a[low];
  do
  {
    while (low < high && a[high].key >= k.key)
    {
      --high;
    }
    if (low < high)
    {
      a[low] = a[high]; ++low;
    }
    while (low < high && a[low].key <= k.key)
    {
      ++low;
    }
    if (low < high)
    {
      a[high] = a[low]; --high;
    }
  } while (low != high);
  a[low] = k;
  return low;
}
```

•
•
•

•
•
• r
  –
  –
  –

MSD

•

LSD

•
•
•

•

-
  -

B B M

B

- B
  -
  -
  -
  -

B

-

1. B key
2.
3.

-
  -
  -
  -
-
-

B

-

1.
2.
3.
   -
   -

M

B+ B+

B+

-
-

-
-
-

B+

-
-
  -
  -

I/O

-
  1. n
  2.

I/O

1.
2.
3.

-
-

1. `buildHeap`
2. `deQuqeue`
3.
   -
   - `deQuqeue`
4. 2 3
5. `buildHeap`

•
•

•
•
•

•

1.
2.

•
•   .
  –
•   .
  –
  –
•   .
  –

•
  –
•

•

•

•

-
-
-
-

-
- G
-

- G G
-
- G

-
-
-

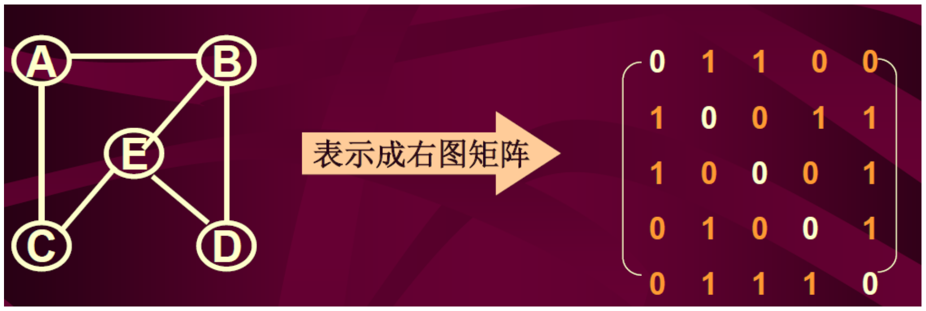
- G G'
- G
-
-
-

-
  -
  -
  -
  -
  -
-
  -
  -
  -

```
template <class TypeOfVer, class TypeOfEdge>
class graph
{
  public:
    virtual void insert(TypeOfVer x, TypeOfVer y, TypeOfEdge w) = 0;
    virtual void remove(TypeOfVer x, TypeOfVer y) = 0;
    virtual bool exist(TypeOfVer x, TypeOfVer y) const = 0;
    int numOfVer() const
    {
      return Vers;
    }
    int numOfEdge() const
    {
      return Edges;
    }

  protected:
    int Vers, Edges;
};
```

-
-
-
-

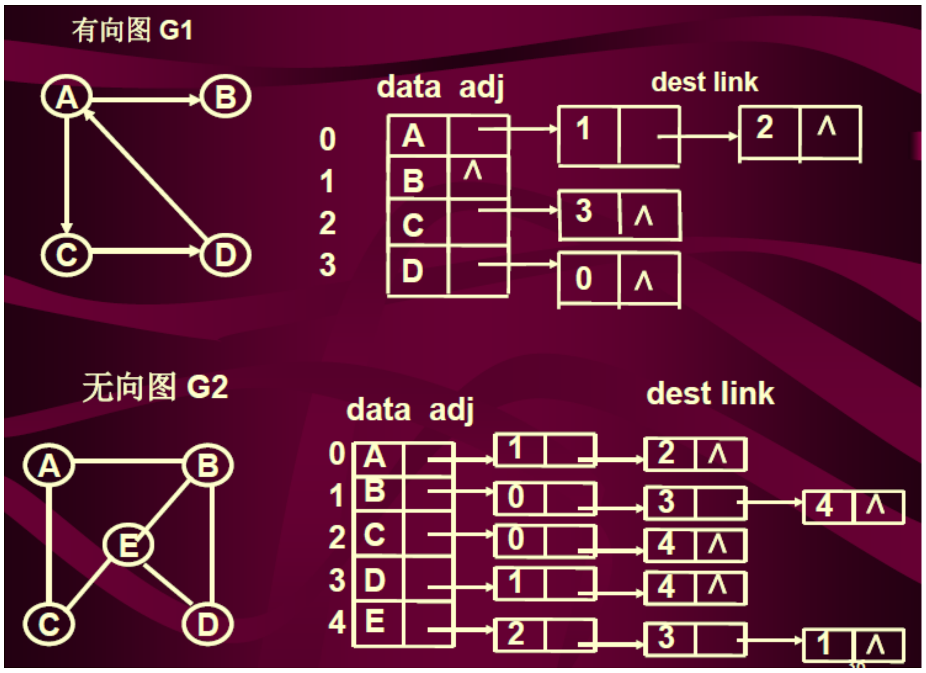


-

•
•

•
• ;
•

•
•
–
–
–
∗
∗
•



有向图 G1
A
B
C
D
data adj
dest link
0
1
2
3
A
B
C
D
∧
1
2
∧
3
∧
0
∧
无向图 G2
A
B
E
C
D
data adj
dest link
0
1
2
3
4
A
B
C
D
E
1
2
∧
0
3
4
∧
0
4
∧
1
4
∧
2
3
1
∧

-
-
  -
-
-
  -
  -
  -

1.
2.
3. ……

DFS

1.
2.
3. ……
4. 2
5.

- dfs

```
void dfs()
{
  visited [v] =false; //

  while(v=    )
  {
    dfs(v,visited);
  }
}
```

- dfs

```
void dfs(v,visited)
{
  visited(v)=true;
  for   v    w
  {
    if(!visited[w])
    {
      dfs(w,visited);
    }
  }
}
```

1.
2.
3.   3
4.   2
5.

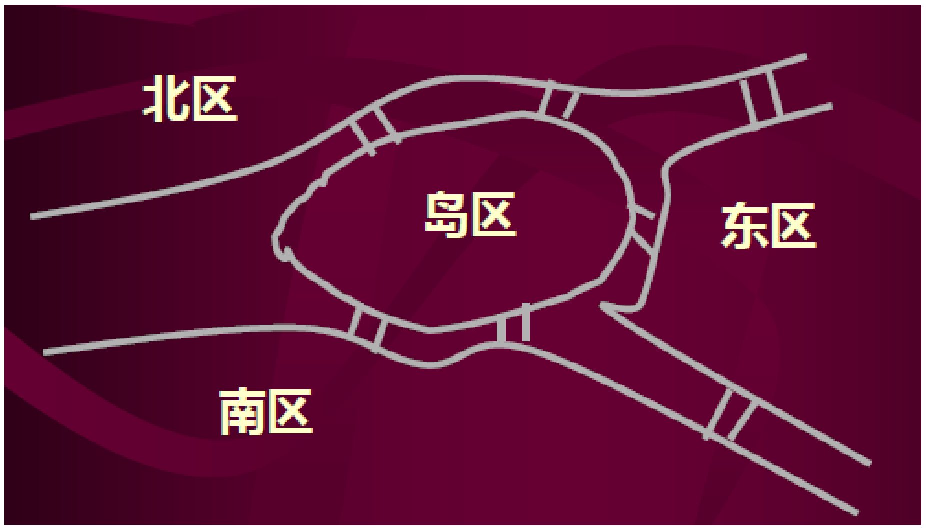
1.
2.
    -
3.

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::bfs() const
{
  bool *visited = new bool[Vers];
  int currentNode;
  linkQueue<int> q;
  edgeNode *p;
  for (int i=0; i < Vers; ++i)
  {
   visited[i] = false;
  }
  cout << "        "<< endl;
  for (i = 0; i < Vers; ++i)
  {
    if (visited[i] == true)
    {
      continue;
    }
    while (!q.isEmpty()) //
    {
      currentNode = q.deQueue();
      if (visited[currentNode] == true)
      {
        continue;
      }
      cout << verList[currentNode].ver <<'\t';
      visited[currentNode] = true;
      p = verList[currentNode].head;
      while (p != NULL)
      {
        if (visited[p->end] == false)
        q.enQueue(p->end);
        p = p->next;
      }
    }
    cout << endl;
  }
}
```

- /
- / / /
-

-
  1.
  2.
  3.
  4.

-



-
-
-

- DFS
-

-
-

Activu on vertex network

-
-
-

AOV

1.
2.

-
-
-

- inDegree
- inDegree
-

```
template <class TypeOfVer, class TypeOfEdge>
void adjListGraph<TypeOfVer, TypeOfEdge>::topSort() const
{
  linkQueue<int> q;
  edgeNode *p;
  int current, *inDegree = new int[Vers];
  for (int i = 0; i < Vers; ++i)
  {
    inDegree[i] = 0;
  }
  for ( i = 0; i < Vers; ++i)
  {
    for (p = verList[i].head; p != NULL; p = p->next)
    {
    ++inDegree[p->end];
    }
  }                                                          //
  for (i = 0; i < Vers; ++i)   if (inDegree[i] == 0) q.enQueue(i);  // 0
  cout << "   " << endl;
  while(!q.isEmpty())
  {
    current = q.deQueue( );
    cout << verList[current].ver << '\t';
    for (p = verList[current].head; p != NULL; p = p->next)
    if( --inDegree[p->end] == 0 )    q.enQueue( p->end );
  }                                                          //
  cout << endl;
}
```

-
-

Activity on Edge

- **AOE**
  -
  -
  -
  -
  -

AOE