Asymptotic Notation

Big O

$O(g(n)) = \{ f(n) \mid \exists c>0, \exists n_0>n, \forall n \geq n_0 : 0 \leq f(n) \leq c*g(n) \}$

Big OMEGA

$\Omega(g(n)) = \{ f(n) \mid \exists c>0, \exists n_0>n, \forall n \geq n_0 : 0 \leq c*g(n) \leq f(n) \}$

Theta

$$\theta(g(n)) = \begin{Bmatrix} f(n) \exists c_1>0, \exists c_2>0, \exists n_0>n \\ \forall n \geq n_0 : 0 \leq c_1*g(n) \leq f(n) \leq c_2*g(n) \end{Bmatrix}$$

Propriedades

$F = O(F) \qquad c*O(F) = O(c*F) = O(F), se\ c>0$

$O(F)+O(G) = O(F+G) = O(max(F+G)), F>0, G>0$

$O(F)*O(G) = O(F*G)$

$O(F)+O(G) = O(F) se\ O(G) < F(N) \forall N > N_0$

Provar Por Recorrência:

Verificar para T(1) e T(n) por recorrência (iterativamente substituindo)

Analise de Algoritmos

simple statements (if, int, return, etc) → O(1)

loop statements (for, while) → somatório do seu primeiro elemento até ao ultimo +1 de tudo o que estiver no seu corpo.

Sorting Algorithms

Compare

	Avg	Worst	Mem	Stable
Insertion	O(n²)	O(N²)	O(1)	Y
Bubble	O(n²)	O(n²)	O(1)	Y
Selection	O(n²)	O(n²)	O(1)	M
Merge	O(nlog(n))	O(nlog(n))	O(n)	Y
Heapsort	O(nlog(n))	O(nlog(n))	O(1)	N
Quicksort	O(nlog(n))	O(n²)	O(log(n))	M

Bubble Sort

```
procedure bubbleSort( A : list of sortable items ) defined as:
  n := length( A )
  do
    swapped := false
    for each i in 0 to n - 1  inclusive do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
        swapped := true
      end if
    end for
    n := n - 1
  while swapped
end procedure
```

Selection Sort

```
void selectionSort(int[] a)
{
for (int i = 0; i < a.length - 1; i++)
{ int min = i; for (int j = i + 1; j < a.length; j++)
   {   if (a[j] < a[min]){min = j;}
}
if (i != min) {int swap = a[i]; a[i] = a[min];a[min] = swap;}}}
```

Insertin Sort

```
insertionSort(array A)
begin
    for i := 1 to length[A]-1 do
    begin
        value := A[i];
        j := i - 1;
        while j >= 0 and A[j] > value do
        begin
            A[j + 1] := A[j];
            j := j - 1;
        end;
        A[j + 1] := value;
    end;
end;
```

QuickSort

```
procedure quicksort(array, left, right)
    if right > left
        select a pivot index (e.g. pivotIndex := (left+right)/2)
        pivotNewIndex := partition(array, left, right, pivotIndex)
        quicksort(array, left, pivotNewIndex - 1)
        quicksort(array, pivotNewIndex + 1, right)

function partition(array, left, right, pivotIndex)
    pivotValue := array[pivotIndex]
    swap array[pivotIndex] and array[right] // Move pivot to end
    storeIndex := left
    for i  from  left to right - 1 // left ≤ i < right
        if array[i] ≤ pivotValue
            swap array[i] and array[storeIndex]
            storeIndex := storeIndex + 1
    swap array[storeIndex] and array[right] // Move pivot to its final place
    return storeIndex
```

Merge Sort

```
function merge_sort(m)
    if length(m) ≤ 1
        return m
    var list left, right, result
    var integer middle = length(m) / 2
    for each x in m up to middle
        add x to left
    for each x in m after middle
        add x to right
    left = merge_sort(left)
    right = merge_sort(right)
    if left.last_item > right.first_item
        result = merge(left, right)
    else
        result = append(left, right)
    return result
function merge(left,right)
    var list result
    while length(left) > 0 and length(right) > 0
        if first(left) ≤ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    end while
    if length(left) > 0
        append left to result
    else
        append right to result
    return result
```

Binary Search

```
    low = 0
    high = N
    while (low < high) {
        mid = low + ((high - low) / 2)
        if (A[mid] < value)
            low = mid + 1;
        else
            //can't be high = mid-1: here A[mid] >= value,
            //so high can't be < mid if A[mid] == value
            high = mid;
    }
    // high == low, using high or low depends on taste
    if ((low < N) && (A[low] == value))
        return low // found
    else
        return -1 // not found
```

Heap

```
* árvore binária (quase) completa
* util para Queues
* raíz A[0]
* Ascendente A[(i-1)/2]
* Descendente E A[2*i+1]
* Descendente D A[2*i+2]
* max-heap A[parent(i)] >= A[i]
* min-heap A[parent(i)] <= A[i]
* Heap-sort = max-heap

public static void heapify(int[] v, int p, int hSize) {
  int l, r, largest;
  l = left(p);
  r = right(p);
  largest=p;
  if(l < hSize && v[l] > v[p]) largest=l;
  if ( r < hSize && v[r] > v[largest]) largest = r;
  if ( largest == p ) return;
  exchange(v, p, largest);
  heapify(v, largest, hSize);
}
public static void exchange(int[] v, int i, int j){
  int tmp = v[i];
  v[i] = v[j];
  v[j] = tmp;
}
static void heapSort(int[] v, int n) {
  buildHeap(v,n);
  for ( int i= n-1; i> 0; --i) {
    exchange(v, 0, i);
    heapify(v, 0, i);
  }
}
static void buildHeap(int[] v,int n){
  int p= parent(n-1);
  for ( ; p >=0 ; --p)
     heapify(v, p, n);
}
static int heapExtractMax(int[] v,  int hSize){
  int max = v[ 0 ];
  v[ 0 ] = v[hSize-1];
  heapify(v, 0, hSize-1);
  return max;
}

static void heapIncreaseKey(int[] v, int i, int key ) {
  v[i] = key;
  while(i>0 && v[i]>v[parent(i)]){
    exchange(v, i, parent(i));
    i = parent(i);
  }
}
static void maxHeapInsert(int[] v, int hSize, int key ) {
  v[hSize] = key;
  heapIncreaseKey(v, hSize, key);
}
```

Listas vs Array

+ Array: Leituras e Escritas
+ Lista: Inserção e Remoção
+ Lista Simples: Inserção e Remoção à cabeça
+ Lista Simples com ponteiro para o último: Inserção ao fim, remoção á cabeça
+ Array: Memória Suficiente
+ Lista: Memória dinamica

Queue (Fifo)

```
public boolean isEmpty() { return tail == tail.next; }
public boolean offer(E e){
  tail.next = new Node<E>( e, tail.next );
  tail = tail.next;
  return true;
}
public E remove() {
  if(isEmpty()) throw new NoSuchElementException();
  Node<E> rem = tail.next.next; tail.next.next = rem.next;
  if ( rem == tail ) tail = tail.next;
  return rem.item;
}
public E poll() { return ( isEmpty() ) ? null : remove(); }
public E peek() { return ( isEmpty()) ? null : tail.next.next.item;}
```

Lista Duplamente Ligada

```
public int size() { return size;}
public boolean isEmpty(){ return head == null;}
public void add(E e) {
  Node<E> n=new Node<E>(e);
  if(head!=null){ n.next=head; head.prev=n;}
  head=n;
}
public boolean remove (E e){
  Node aux=search(e);
  if(aux!=null){
    if(aux.prev != null) {aux.prev.next = aux.next;}else{head = aux.next;}
    if(aux.next != null){ aux.next.prev = aux.prev;}
    return true;
  }
  else return false;
}
protected Node<E> search( E e ){
  Node<E> aux = head;
  while( aux != null && !aux.key.equals(e))
    aux=aux.next;
  return aux;
}
public static <E> void show(DNode<E> l){
  System.out.print("< ");
  while(l != null){ System.out.print(l.key+" "); l = l.next;}
  System.out.println(">");
}
public void reverse() {
  Node<E> iter = head, aux;
  while( iter != null ) {
    head = iter;
    aux = iter.next;
    iter.next = iter.prev;
    iter.prev = aux;
    iter = aux;
  }
}
```

HashTables

+ Endereçamento Directo
+ Associam chaves a valores
+ Tabela Base de Indexação
+ Resolução de Colisões
+ Dispersão por Indicies Livres
+ Dispersão por separação em Listas
+ Função de Dispersão (k%M, M tamanho do array, M primo ou impar)
+ Load factor before reash (70%)

Arvores BST's

Varrimento

+ Prefixo : Node → Left → Right
+ Infixo: Left → Node → Right
+ Pósfixo Left → Right → Node

Árvore Binária de Pesquisa

+ Cada Node tem uma chave associada
+ Cada chave associada a um Node é maior que todos os Nodes à esquerda
+ Cada chave associada a um Node é menor que todos os Nodes à direita

Tree-Search

```
public Node<E> search(E e){
  return recursiveSearch(root,e);
}
private static <E extends Comparable<E>>
Node<E> recursiveSearch(Node<E> h,E x){
  if(h==null || equals(x,h.key)) return h;
  if(less(x, h.key)) return searchR(h.left,x);
  else return searchR(h.right,x);
}
```

ITERATIVE-TREE-SEARCH (x,k)

```
while  x<> NIL and k<>key[x]
  do if k<key[x]
       then x ←  left[x]
       else x ←  right[x]
return x
```

TREE-SUCESSOR(X)

```
if right[x] <> NIL
    then return TREE-Minimum(right[x])
y ← p[x]
while y <> NIL and x right [y]
    do x ← y
       y ← p[y]
return y
```

TREE-INSERT(T,z)

```
y ← NIL
x ← root[T]
while x <> NIL
   do y ← x
      if key[z] < key [x]
         then x ← left[x]
         else x ← right[x]
p[z] ← y
if y = NIL
   the root[T] ← z
   else if key[z] < key[y]
         then left[y] ← z
         else right[y] ← z
```

Árvores Balanceadas

```
private Node rotRight(Node x)
{Node aux=x.left; x.left=aux.right; aux.right=x; return aux;}

private Node rotLeft(Node y)
{Node aux=y.right; y.right=aux.left; aux.left=y; return aux;}
```

Rotação Simples Esquerda/Direita

+ Se a altura da sub-arvore da direita for superior a 1 do que da árvore da esquerda e se e só se a altura da sua sub-árvore for superior ou igual.
+ Se no caso anterior a sua altura for inferior, é necessário uma rotação dupla.

Procura Externa: B-TREES

+ Armazenamento externo
+ grau **t**
+ Todas as folhas tem altura **h**
+ Cada nó (excepto raiz) tem pelo menos **t-1** chaves e no máximo **2*t-1** chaves
+ A inserção de k é sempre numa página folha y.

Se a folha y onde queremos inserir k estiver cheia, dividimos (split) a folha y, à volta da chave do meio (keyt), em duas páginas que contêm (t-1) chaves. A chave do meio é inserida na página parent de y.

Se o parent de y estiver cheio, também tem que ser dividido para que a chave seja inserida, e a propagação de divisões poderá ser até a raiz.