Aula prática

① nº chamadas recursivas $(nc(n))$

$nc(n)$ { 0 $n=1$; $1+nc(\frac{n}{2})$ $n>2$ par ; $1+nc(\frac{n}{2})$ $n>3$ ímpar

Não interessa se é par ou ímpar, porisso

$nc(n)$ { 0 $n=1$; $1+nc(\frac{n}{2})$ $n>2$ par.

Temos de utilizar R2a ou R2b

$Nc(n) = O(\log n)$

Atendendo à Regra R2a, a condição sai satisfeita, como queremos que $b=1$ então.

②

$nc(n)$ { 0 $n=1$ (1 par) ; $1+Nc(\frac{n}{2})+1+Nc(\frac{n}{2})$ $n>2$

$= 2\,Nc(\frac{n}{2})+2$

$Nc(n) = O(n)$

AED prácticas 21-9-2010

R1 $$T(n) = \begin{cases} a & n = 0 \quad 1 \\ b\,T(n-1) + c & n \geq 1 \quad 2 \end{cases}$$

$a \geq 0 \quad b \geq 1 \quad c \geq 1$

$$T n = \begin{cases} O(n) & b = 1 \\ O(b^n) & b > 1 \end{cases}$$

R2 (a)

$$T(n) = \begin{cases} a & n = 0 \quad 1 \\ b\,T\left(\frac{n}{2}\right) + O(1) & n \geq 1 \quad 2 \end{cases}$$

$a \geq 0 \quad b = 1, 2$

$$T(n) = \begin{cases} O(\log n) & b = 1 \\ O(n) & b = 2 \end{cases}$$

R2 (b)

$$T(n) = \begin{cases} a & n = 0 \quad 1 \\ b\,T\left(\frac{n}{c}\right) + O(n) & n \geq 1 \quad 2 \end{cases}$$

$a \geq 0 \quad b \geq 1 \quad c \geq 1$

$$T(n) = \begin{cases} O(n) & b < c \\ O(n \log_c n) & b = c \\ O(n^{\log_c b}) & b > c \end{cases}$$

(3)

$$NC(n) \begin{cases} 0 & n=1 \\ 1+NC(n-1)+1+NC(n-1) & n \geq 2 \end{cases}$$

$$(=) NC(n) \begin{cases} 0 & n=1 \\ 2NC(n-1)+2 & n \geq 2 \end{cases}$$

R1 - $a=0$ $b=2$ $c=2$

$NC(n) = O(2^n)$

(4)

$$NC(n) \begin{cases} 0 & n=2 \\ 1+NC\left(\frac{n}{2}\right)+1+NC\left(\frac{n}{2}\right) & n \geq 3 \end{cases} = \begin{cases} 0 \\ 4NC\left(\frac{n}{2}\right)+4 \end{cases}$$

$+1+NC\left(\frac{n}{2}\right)+1+NC\left(\frac{n}{2}\right)$

$O = (b^n)$

$$e(n) = \begin{cases} 1 & n=1 \\ 2e\left(\frac{n}{2}\right) + O(n) + d & n \geq 2 \end{cases}$$

→ Obriga a utilizar recorrência 2ª

d: constante (nº de passos efectuados, 5')

$e(n)$: nº de passos executados por ff+/s quando o vector tem n elementos

$O(n \log n)$

FFT (vec)
remark $O(n)$
fft3 $O(n \log n)$

$O(n \log n)$

ege Sotw Recursivo

$$C(n) \begin{cases} 0 & n = 0,1 \\ 2C\left(\frac{n}{2}\right) + F(n) & n \geq 2 \end{cases}$$

$$F(n) = \begin{cases} \frac{n}{2} & \text{melhor caso} \\ n-1 & \text{pior caso} \end{cases}$$

```
if (pair1 == pair2)
{ pair1 = pair2; iter1 = iter2; }
while (pair1 != null)
{   res.addLast(new PairClass(pair1.getFirst(), pair1.getSecond()))
    pair1 = this.nextPair(iter1)
}
Return new PolynomialCl(Res);
```

(9) Último

Construtor

```
private PolynomialCl(List<Pair<I,I>> theList){
    list = theList;
}
```

Complexidade

No mínimo entra o nº de vezes do maior pequeno

No máximo max m+n exemplo $a + a^3 + a^5$ Comparar 1 do $a^2 + a^4 + a^6$ cada vez, 1º um de cima depois um de baixo...

max m n

$O(m+n)$

$O(\max(m,n))$

prática 2

5) public static int LCS (char [] seq1, char [] seq2)

int [][] memory = new int [seq1.length+1] [seq2.length+1];

inicializa posições com -1

return LCSMem(memory, seq1, seq2, seq1.length, s2.length)

private static int LCSMem (int [][] memory, char [] seq1, char [] seq2, int length1, int length2)

if (memory [length1] [length2] == -1)

if (length1 == 0 || length2 == 0)

memory [length1] [length2] = 0

else if (seq1 [length1 - 1] == seq2 [length2 - 1])

memory [length1] [length2] = 1 + LCS (memory, seq1, seq2 length1-1, length2-1);

else

Math.max (LCSMem (memory, seq1, seq2, length1, length2-1), LCSMem (memory, seq1, seq2, length1-1, length2))

}

Return memory [length1] [length2];

n n

n x m

complexidade espacial - se m for o comprimento da 1ª coluna e da segunda, a complexidade espacial é m x m

complexidade temporal

O custo temporal pode ser ~~mesmo~~ menor do que $\leq O(2 \times m \times m$

$O(m \times m)$

6

```
public static double fun (int n){
    double memory[] = new double [n+1];
    return funRecMem (memory, n)
}
Inicialização

private static funRecMem (double memory, int n){
    if (memory [n] == -1)
        if (n<3)
            memory [n] = n
        else
            memory [n] = Math.sqrt (n) + funRec (n-1) * funRec (n-3);
    return memory [n]
}
```

Complexidade espacial n

Complexidade temporal

Cada posição de memória é calculada 0 ou 1 vez

Cada " dá origem a 0 ou 2 chamadas recursivas, ou seja no máximo n x 2

```
public static long nomeMetodo (int n) throws NegativeNumberException
    long[] memory = new long [n+1] ---> inicializa (memory);
    if (n<0)
        throw new NegativeNumberException
    else
        return funcRec(n)
}

public void inicializa (long[] mem){
    for (int i=0; i<mem; i++)
        mem[i] = -1;
}

private static long nomeRec (int n){ ---> if (memory[n] == -1){
    if (n==0 || n==1){
        memory[0] = 2 * n+1
        memory[1] = memory[0]
    else{
        long prod=1
        for (int i=0; i<n/2; i++){
            if (memory[n-i-1] == -1)
                memoria[i] = nomeRec(n-i-1);
            if (memory[i] == -1)
                memoria[i] = nomeRec(i)
        memory[n] = prod * (memory[n-i-1] % memory[i] +1);
    }
    return memory[n];
```

a complexidade espacial é n

a complexidade temporal é

não percorrer lista com geT's
mas sim com o iterador

AED
Aula 3

(8)

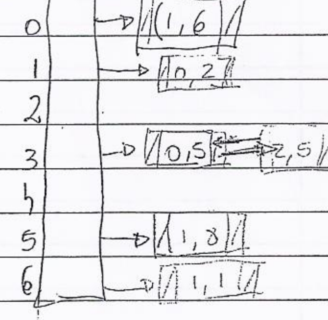
```
public SparseMatrix <E> transpose ()
{
  List <Entry <Integer, E>> [] lineAux = *
  for (int i=0; i < numberOfColums; i++)
  {
     lineAux [i] = New DoubleLinkedList < Entry <integer, E>>();
  }
  for (int i=0; i < lines.length ; i++)
  {
     Iterator <Entry <Integer, E> it =
        line [i]. iterator();
     while (it.hasNext())
     { this.add (i, iter, lineAux);
     // Entry <Integer, E> e1 = New EntryClass();
     // e1 = it.next();
     // lineAux [e.getKey()].add(i, e1.getValue());
     }
  return New SparseMatrix<E> (lineAux,
                               lines.length);
}

* = (List <Entry <Integer, E>>[]) New List [numberOfColums];

private SparseMatrix ( List <Entry <Integer, E>>
            theArray int theNumbOfColumns)
{ lines = theArray ; numberOfCollums = theNumberOfColums; }
```

M =

0	2	0	1	0	0	0
6	0	0	0	0	8	1
0	0	0	5	0	0	0




```
private E Void ...lines (int i, Iterator it
     List <Entry <Integer, E>> lineAux)
{ while (it.hasNext())    it.Next(); E element = entry.getValue();
  { Entry <Integer, E> e = New EntryClass <Integer, E>
                                          (i, element)
    res[iter.Next().getKey()].addLast( e);
```

```
private void addLine (int line, Iterator<Entry<Integer, E>>...
                      iter, List<Entry<Integer, E>>res)

{   while (iter.hasNext())
  } Entry<Integer, E> entry = iter.next()
    int column = entry.getKey();
    E element = entry.getValue();
    Entry<Integer, E> newEntry =
        New EntryClass<Integer, E>(line, element);
    res[column].addLast(newEntry);
{
```

$mn \rightarrow$ n: linhas

$m \rightarrow$ n: colunas

$k \rightarrow$ n: elementos diferentes de zero

$mn + n + k$

```
public Polynomial sum (Polynomial poly)
{ Iterator <Pair<Integer, Integer>> iter1 = list.iterator();
                                    iter2 = poly.list.iterator();
  List<Pair<Integer, Integer>> res = new DoubleLinkedList<Pair<Integer,
                                                              Integer>>();
  Pair<Integer, Integer> pair1 = this.nextPair (iter1);
  Pair<Integer, Integer> pair2 = this.nextPair (iter2);
  while (pair1 != null && pair2 != null)
  {   int degree1 = pair1.getFirst();
      int degree2 = pair2.getFirst();
      if (degree1 == degree2)
      {   int value = pair1.getSecond() + pair2.getSecond();
          if (value != 0)
              Res.addLast (newPairClass (degree1, value);
          pair1 = this.nextPair (iter1);
          pair2 = this.nextPair (iter2);
      }
      else if (degree1 < degree2)
      {   res.addLast (newPairClass (degree1, pair1.getSecond()));
          pair1 = this.nextPair (iter1);
      }
      else
      { Res.addLast(newPairClass (degree2, pair2.getSecond()));
        pair2 = this.nextPair (iter2);
      }
{


Private Pair<Integer, Integer> nextPair (Iterator<Pair<I, I>> it)
{- if (it.hasNext())
       Return it.next();
   else
       Return null;
}
```

* continues

4) Private static long noNameRec(long[] memory, int n){

```
if (memory[n] == -1)
    if (n == 0 || n == 1)
        memory[n] = 2 * n + 1;
    else {
        long prod = 1;
        for (int i = 0; i < n/2; i++)
            prod = prod * noNameRec(memory, n - i - 1) %
            noNameRec(memory, i) + 1);
        memory[n] = prod;
    }
return memory[n];
```

complexidade espacial O(n)

Init O(n)
Calculo O(n²)

O(n²)

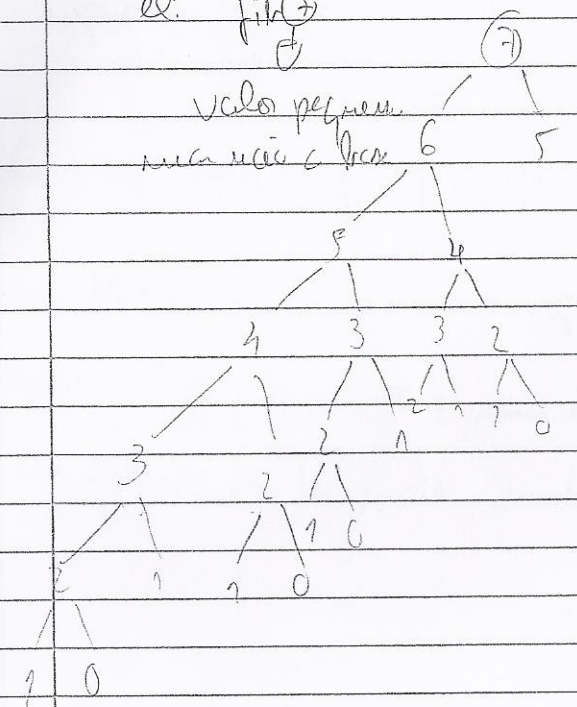
Exame tem 1 pag · Calcule complexidade
· aplique a técnica memo.

AGD

Custo algoritmo recursivo

- calcular nº chamadas recursivas
- verificar se é recursividade 1 ou 2
- se não for complexidade de 1 ou 2 fazer manualmente

ex: fib(7)

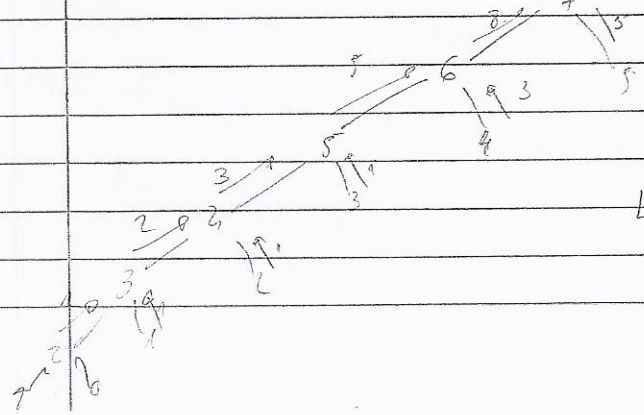


como tem chamadas recursivas repetidas é complexidade aproximada, exponencial pode!

para evitar o problema anterior temos de utilizar a forma memoização

1º declarar todas as chamadas recursivas
2º inicializar com valores negativos

a forma memoização - vai ao array ver se já calculado (vê inicializado ou array), e faz o seguinte



0	1	1	2	3	5	8	13
0	1	2	3	4	5	6	7

Complexidade temporal

inicialização	$O(n)$
cálculo	$O(n)$
total	$O(n)$

No java até é possível declarar arrays de tipos que não sejam os tipos básicos. para contornar esta situação temos de fazer o seguinte

```
array = (E[]) new Object[capacity]
```

este código gera um warning, mas não tem problemas

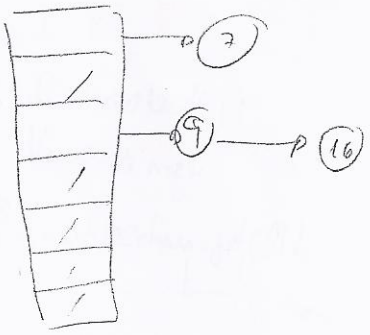
tabela de dispersão

```
public static int hash(String key) {
int a = 127  // numero primo
int b = 2147483647  // numero primo
int hashcode = 0
for
    hashcode = (hashcode * a + key.charAt( ));
```

a tabela de dispersão resolve o problema de colisões.

lide do

new - ordDoubleLL → é para implementar Dicionário ordenado, ou seja tenho de ter mais 2 métodos, getHead, getTail.



7 % 7 - 0 7 está no zero porque 7 % 7 dá zero.

Quando a tabela enche a única solução é aumentar o tamanho (dobrar o tamanho) da tabela de dispersão. Atenção: Não se aproveitam os Dicionários, ou seja, temos de calcular toda a posição novamente

É necessário fazer o código usado anteriormente. (desperdício)

Iterador 3 métodos

- tem seguinte
- Qual o seguinte
- rewind

Guardar num estado que quando pedirem o próximo é imediato.

Node > nextToReturn

de acordo com a condição como o construtor fica

```
construtor (head) →
    firstNode = head;
    this.Rewind();
hasNext
    nextReturn != Null
```

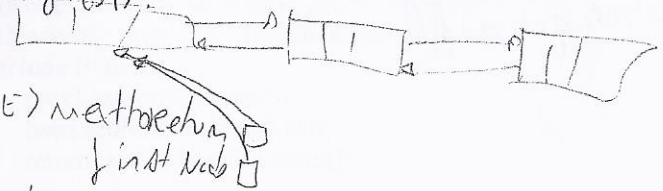
Rewind (temos de guardar o primeiro variável)

~~DLN<E> firstNode~~

```
next toReturn = firstNode
```

E Next

```
if(!thishasNext())
    throws new...
else
el = nextToReturn.getEl();
```



DLN<E> nextToReturn

firstNode

```
nextToRead = nextToRead.getNext;
Return el;
```

Lista duplamente ligada.

```
DLN<E> nextToReturn
// firstNode
// lastNode

Construtor (head, tail)
    firstNode = head
    lastNode = tail
    this.rewind();

hasNext
    nextOR != null

rewind
    nextToReturn = firstNode
    prevToReturn = null

E next()
    if (!this.hasNext())
        throws new...

E el = nextToReturn.getE();
prevToReturn = nextToReturn.getPrev();
nextToReturn = nextOR.getNext();
Return el;

fullForward
    nfn = null
    pfn = lastN
```

Ex 10

```
protected void restoreAllPreviousPointers(){
        DListNode<E> node = head;
        DListNode<E> prevNode = null;

        while(node!=null){
                node.setPrevious(prevNode);
                prevNode = node;
                node = node.getPrevious();
        }
}
```

Complexidade Temporal
O(n) em todos os casos

Ex 11

```
public void addLast (E element){
        if(head == null){
                DListNode<E> el = new DListNode<E>(element);
                el.setPrevious(el);
                el.setNext(el);
                head = el;
        }
        else{
                DListNode<E> tail = head.getPrevious();
                DListNode<E> el = new DListNode<E>(element, tail, head);
                tail.setNext(el);
                head.setPrevious(el);
        }
}
```

Complexidade Temporal
O(1) em todos os casos

```
public E removeLast() throws ...{
        if(head == null)
                throw new ...
        DListNode<E> lastNode = head.getPrevious();
        DListNode<E> prevLast = lastNode.getPrevious();
        if(lastNode != head){
                prevLast.setNext(head);
                head.setPrevious(prevLast);
                return lastNode.getElement();
        }
        else{
                head = null;
                return lastNode.getElement();
        }
}
```

Complexidade Temporal
O(1) em todos os casos

Ex 12

```
public void insert (E element) throws .... {
        if(head == null)
                head = new SListNode(element);
        else{
                SListNode<E> curr = head;
                SListNode<E> prev = null;
                while(curr != null) {
                        if(curr.getElement().compareTo(element) == 0)
                                throw new ....
                        else if(curr.getElement().compareTo(element)>0){
                                SListNode<E> el = new SListNode<E>(element, curr);
                                if(prev == null)
                                        head = el;
                                else
                                        prev.setNext(el);
                                break;
                        }
                        prev = curr;
                        curr = curr.getNext();
                }
                if(curr == null){
                        SListNode<E> el = new SListNode<E>(element,null);
                        prev.setNext(el);
                }
        }
}
```

Complexidade Temporal
O(n) em todos os casos

AED)

(12) public void insert (E element) throws RepeatedElementExc
{

```
SList.Node<E> prevNode = null;
SListNode<E> currNode = head;
while (currNode != null && currNode.getElement().compareTo(element) < 0)
{ prevNode = currNode;
  currNode = currNode.getNext();
}
if (currentNode != null && currentNode.getElement().compareTo(element) == 0) {
  throw new RepeatedElementExc();
SListNode<E> newNode = new SListNode<E>(element, currNode);
if (prevNode == Null)
   head = newNode;
else
   prevNode.setNext(newNode);
}
```

→ não sabemos o peso no entanto o enunciado diz para considerar constante

custo

no melhor caso - tudo é constante menos o nº de vezes que entramos no while O(1)

pior caso O(n) - percorrer todos os nós para inserir no cauda ou maior que todos os elementos

caso esperado O(n) - temos de percorrer metade dos elementos da lista

Atenção o(1) - quando a lista está vazia é errado

(13) ——— P utilizando Parâmetros de acumulação

```
public int[] static(){
  int[] res = new int[3];
  this.compStat(root, res);
  return res;
}
```

Se o retorno for um objecto, este método é mais simples, no outro caso é mais simples para tipos básicos (char, int...)

```
private void compStat(BSINode<K,V> node, int[] res)
{
  if (node != null)
  {
    if (node.getLeft() == null)
      if (node.getRight() == null)
        res[0]++;
      else
        res[1]++;
    else
      if (node.getRight() == null)
        res[1]++;
      else
        res[2]++;
    this.compStat(node.getLeft, res);
    this.compStat(node.getRight, res);
```

Nota: podíamos fazer o teste com o método isLeaf.

(13)

```
public int[] statistics()
{
    Return this.compStats(root);
```

```
Private int[] compStats(BstNode<K,V> node)
{ if(node == null)
    return new int[3];
  else
  { int[] res = new int[3];
    int[] resLeft = this.compStats(node.getLeft());
    int[] resRight = this.compStats(node.getRight());

    for(int i=0; i<res.length; i++)
      res[i] = ResLeft[i] + ResRight[i]
    Return res;
}
```

complexidade n (n - número de nós da árvore), tem de percorrer todos os nós da árvore.

O nº de execuções do compStats é $2n+1$ sempre

(4) public int pairOfSiblingsWithTheSameValue () {

return this.Pairs (root);

}

private int ... Pairs (BstNode<K,V> node) {

if (node == null)

return 0;

else {

BstNode<K,V> left = node.getLeft();

BstNode<K,V> Right = node.getRight();

int leftResult = this.Pairs(left);

int RightResult = this.Pairs(Right);

if (left != null && right != null && left.getValue().equals(Right.getValue()))

Return 1 + leftResult + RightResult;

else

Return leftResult + RightResult;

}

}

(15)

```
public BinarySearchTree(BinarySearchTree<K,V> tree)
    {
        root=this.clone(tree.root);
        currentSize = tree.currentSize();
    }

Private BstNode<K,V> clone(BstNode<K,V> node)
{
    if(node == null)
        return null;
    else
    { EntryClass<K,V> entryC = node.getEntry().clone
      BstNode<K,V> leftC = this.clone(node.getLeft());
      BstNode<K,V> RightC = this.clone(node.getRight());
      BstNode<K,V> nodeC = new BSTNode<K,V>(entryC, leftC, RightC);
      return nodeC;
    }
}
```

A complexidade é sempre n, ou seja o nº de nós da árvore, porque é sempre necessário percorrer todos os nós.

16

```
public int getDegree() throws EmptyTreeException {
    if (root == null)
        throw new EmptyTreeException();
    return this.getDegree(root);
}

private int getDegree(TreeNode<E> node) {
    int nodeDegree = node.getNumberOfChildren();
    int maxDegree = nodeDegree;

    for (int i = 0; i < nodeDegree; i++) {
        int childDegree = this.getDegree(node.getChild(i));
        if (childDegree > maxDegree)
            maxDegree = childDegree;
    }

    return maxDegree;
}
```

O(n)

```
public int[] levelSizes() throws EmptyTreeException
{
if (root == null)
  throw EmptyTreeException();
return this.levelSizes(root, 0);

private int[] levelSizes(TreeNode<E> node, int level)
{  int[] res = new int [currentHeight];
   Res[level] = 1;
   TreeNode<E> child = node.getChild();
   while (child != null)
   {
       int[] childRes = this.levelSizes(child, level+1)
       for (int i=0, i<currentHeight, i++)
           Res[i] += childRes[i];
       child = child.getSibling();
   }
  return res;
}
```

AED 16-10-2010

(18) - As estruturas de dados mais adequadas para implementar a interface QueueWithRanks são uma AVL ou uma árvore vermelha e preta, e TAD mais adequada, para implementar uma fila, é uma lista duplamente ligada com cabeça e cauda.

AVL / RB (R, DoublyLinkedList(E))

enqueue

	M.C	P.C	C.E	
find (Rank)	1	log R	log R	
se ∃, insere DLL (cauda)	1	1	1	1º caso
se ∄ cria DLL	1	1	1	
insert DLL (cauda)	1	1	1	1º caso
insert (R, DLL) (cauda)	log (R)	log (R)	log (R)	
total	1	log (R)	log (R)	

R → altura da árvore, ou seja, o nº de categorias na fila com cat

O elemento não existe, temos de inserir

Nota: no pc e no CE escolhemos sempre a complexidade + elevada, por exemplo caso 1 < log (R)

dequeue (rank)

	MC	PC	CE
find (Rank)	1	log R	log R
se ∃, remove da cabeça da lista	1	1	1
se lista vazia (encontra e remove) remove(rank)	log R	log R	log R
se ∄, lança exceção	1	1	1
total	1	log R	log R

melhor caso - encontrar na 1ª posição, remove e a lista não fica vazia

dequeue()

	MC	PC	CG
se árv. vazia lança exepção	1	1	1
se não • maxentry	logR	logR	logR
remoção à cabeça da lista	1	1	1
• se lista vazia remoção da entrada	log R	logR	logR
total	log R	log R	log R

---

(19) F Fila1 Fila<E> - DLL<E>

T Fila2 Fila<E> - DLL<E>

filaGlobal Fila(Boolean) DLL(Bool)

Inserir na cauda da DLL 1

Inserir na cauda da DLL1

Retira

se vazia lança exepção

se F removemos e retornamos o que se encontra à cabeça da fila 1

se V " " " " " " " " da fila 2

Iterador de nível 1 retornamos iterador fila 1

" " " 2 " " " 2

[obter iterador

[iterar tudo

Dic (Representante, Dic (G,E)

numSize - o size do dicionário, devolve o nº de famílias, preciso mai funções, tenho de fazer um contador

TD (Representante, [TD / AVL / RB] (G,E)

→ qualquer uma das 3 é aceitável, a tabela de dispersão tem de ter um pré-dimensionamento k, como mai sistema é necessário fazer rehash, então AVL e RB também é aceitável

a) Hashtable <e-mail, visitante> visitantes

b) Hashtable <tipo, nome> Actividades

c) Hashtable <e-mail, boolean> entrada/saída

d) Hashtable <data, Hasmap <e-mail, nome>> ??? participações

e) Hashtable <email, data> participações

f) Hashtable <email, boolean> entrada/saída

1º Rascunho

a) TD (tabela de dispersão)

TD <e-mail, visitante> visitantes

TD <nome, Actividade> actividades

Dic - TD

DicOrd - RB-AVL

Actividade

RB/AVL <data, nº participantes> - h | Dispersão → DLL (data, nº part)

tipo

nome

Visitante

TD <nome, nome> participações Actuais

bool naPraça

RB/AVL <tipo, AVL/RB <nome, DicOrd <data-inic, nº participações>>> - g

DLL (data-ini, nº participações)

Dispersão

para simplificar, podemos utilizar chave composta

DicOrd (tipo, nome), DLL (data-ini, nº participações)

custos

a) pesquisa | 1
ciclo visitante | Nº atividades simultâneas que pode
| Nº atividades sim... porque é > 1

↳ para cada atividade tem de ciclo numa lista para cada participação, mas é constante

b) O(1)

c) O(1)

d) ver se visitante existe O(1)
" " atividade " O(1)
" " está no parque O(1)
⋮

log(N) + log(NT) + log(DA) + log(DA participante)
Data atividade

e) O(1)

f)
verificar Size	O(1)
Boleano	O(1)
total	O(1)

g) O(participações total do visitante pedido)

h) O(ordem de nº participações totais na atividade)

↳ percorrer AVL/RB Atividades

representante — podemos ser mesmo aqui — não interessa muito

Exo - HashTable<R, HashTable<E, ?>>

int currentSize;

size → retorna o currentSize → O(1)

Contains /1

operações	emC	∧C	CE
obtem o Representante	O(1)	O(1)	O(1)
pesquisa por representante	O(1)	O(1)	O(1)
pesquisa no resultado	O(1)	O(1)	O(1)
total	O(1)	O(1)	O(1)

insert /1

operação	emC	∧C	CE
contains	O(1)	O(1)	O(1) → obtem Rep O(1) ...
existe throws exc	O(1)	O(1)	O(1)
se existe na TD throws TD	O(1)	O(1)	O(1)
existe pesquisa	O(1)	O(1)	O(1)
cria TD interna	O(1)	O(1)	O(1)
insere TD II	O(1)	O(1)	O(1)
insere TD I	O(1)	O(1)	O(1)
inc size	O(1)	O(1)	O(1)
total	O(1)	O(1)	O(1)

remove /1

operação	emC	∧C	CE
obt Rep	O(1)	O(1)	O(1)
contains existe remove TDII	O(1)	O(1)	O(1)

21- HashTable<V, RB<K,K>>

AVL

MinHeap<K,V>

size → O(1) em todos os casos (pode o size do MinHeap)

minEntry/0

Operação	mc	ac	ce
Se fila vazia Exc	O(1)	O(1)	O(1)
Retorno do minKey	O(1)	O(1)	O(1)

insert/2

Operação	mc	ac	ce
pesquisa na HashTable	O(1)	O(1)	O(1)
não existe cria RB	O(1)	O(1)	O(1)
insere na RB	O(1)	O(1)	O(1)
insere na Hash	O(1)	O(1)	O(1) → medio
insere na Heap	O(1)	O(log m)	O(log m)
existe compara com o mes	O(log m)	O(log m)	O(log m)
ele tem inserir insere na RB	O(1)	O(log m)	O(log m)
insere na heap	O(log K)	O(log K)	O(log K) → medio entradas na

Total: O(log K)

↳ K > m

removeMin/0

Operação	mc	ac	ce
vazia Exception	O(1)	O(1)	O(1)
remover da Heap	O(1)	O(log m)	O(log m)
pesquisa na hash	O(1)	O(1)	O(1)
remover na árvore	O(log K)	O(log K)	O(log K)
Apagar entrada Hash	O(1)	O(1)	O(1)

total → O(m)

↳ m > K

dequeue/1 — Seja n o nº de Ranks

Operações	melhor caso	pior caso	caso esperado
pesquisa (Rank)	O(1)	O(log n)	O(log n)
se existe remove à cabeça	O(1)	O(1)	O(1)
se lista vazia remove rank	O(log n)	O(log n)	O(log n)
se não existe lança exceção	O(1)	O(1)	O(1)
Total	O(1)	O(log n)	O(log n)

↑ (melhor caso) encontra logo e ao remover a lista continua com elementos

dequeue/0 — seja n o nº de ranks

operações	melhor caso	pior caso	caso esperado
if vazia → exception	O(1)	O(1)	O(1)
else o max entry	O(log n)	O(log n)	O(log n)
remoção à cabeça	O(1)	O(1)	O(1)
se lista vazia remove rank	O(log n)	O(log n)	O(log n)
Total	O(log n)	O(log n)	O(log n)

14 - DoublyLinkedList<E> level1

DoublyLinkedList<E> level2 "

DoublyLinkedList<Boolean> //True → 2, false → 1

enqueue level 1 / 1

Operação	melhor caso	Pior caso	Esperado
Inserir à cauda da lista 1	O(1)	O(1)	O(1)
Inserir falso na lista de booleanos à cauda	O(1)	O(1)	O(1)

enqueue level 2 / 1

Operações	melhor caso	pior caso	esperado
inserir à cauda da 2	O(1)	O(1)	O(1)
inserir True para cauda a	O(1)	O(1)	O(1)

dequeue

operação	mc	ac	cc
remover à coleção de booleanos	O(1)	O(1)	O(1)
se for True remover à coleção da 2 se for False remover à coleção da 1	O(1)	O(1)	O(1)

Iteradores → O(1) para os dois o retorno

O(n) para iterar com cada um

↓

nº de elementos da lista

22- HashTable<V, Integer>
RB<K, V>

entries with value /1

Operação	amc	pc	cp
pesquisa de valor	O(1)	O(1)	O(1)
existe → ret ind	O(1)	O(1)	O(1)
!existe → ret 0	O(1)	O(1)	O(1)
total	O(1)	O(1)	O(1)

minEntry → O(log n) da árvore
maxEntry → O(log n) da árvore
↳ nós elementos

isEmpty → O(1) → operações da árvore
size() → O(1)
iterador() → O(n)
find /1 → O(log n) → operação da árvore

insert /2

Operação	amc	pc	cp
insere na RB	O(log n)	O(log n)	O(log n)
Ret null pesquisa na TD	O(1)	O(1)	O(1)
existe incrementa	O(1)	O(1)	O(1)
!existe cria entrada	O(1)	O(1)	O(1)
ret val ≠ val-novo pesquisa TD val	O(1)	O(1)	O(1)
dec ind	O(1)	O(1)	O(1)
val=0 remove	O(1)	O(1)	O(1)
pesquisa val-novo	O(1)	O(1)	O(1)
existe inc ind	O(1)	O(1)	O(1)
!existe cria ret val = val - novo mod mexe	O(1)	O(1)	O(1)
Total	O(log n)	O(log n)	O(log n)

Remoção → O(log n) → remove da RB, o resto é constante

23- RB<E, Integer>	DLL<E> elementos
DLL<E>	DLL<E> máximos
E max	E max
O(log n) no pushback	O(1) em todos os casos

AED 24

```
Public Entry<K,V> 2ndMinEntry() throws NoSuchElementExec
{
    if(currentSize<2)
        throw new NoSuchElementExec();

    if(currentSize==2 || array[1].getKey().compareTo(array[2].getKey())<0)
        Return array[1];
    else
        Return array[2];
}
```

Complexidade constante O(1)

9. AED 30-11-2010

TD < Automoveis, Automoveis > maximo eficiencia

AUL/RB < Marca, Automoveis > modelo Marca

TD < Marca modelo, Automoveis > Automoveis

Automovel

- marca
- modelo
- int capacidade
- int autonomia

custo

a)

$$mc(n) \begin{cases} 0 & n=1 \\ 1+mc\left(\frac{n}{2}\right) & \text{par} \\ 1+mc\left(\frac{n}{2}\right) & \text{impar} \end{cases} = \begin{cases} 0 & n=1 \\ 1+mc\left(\frac{n}{2}\right) & n \geq 2 \end{cases}$$

$$mc(n) \begin{cases} 0 & n=1 \\ 1+mc\left(\frac{n}{2}\right)+1+mc\left(\frac{n}{2}\right) & n \geq 2 \end{cases} = \begin{cases} 0 & \\ 2+mc\left(\frac{n}{2}\right) & mc(n) = O(n) \end{cases}$$

0 $n=1$