ESTRUTURAS DISCRETAS

Ana Neri (ana.i.neri@inesctec.pt)

DI,
Universidade do Minho

17 de Abril 2023

Conteúdo

Conjuntos e Multi-conjuntos

O conjunto tem como operações fundamentais:

- procura
- e inserção

e ainda é comum terem as operações:

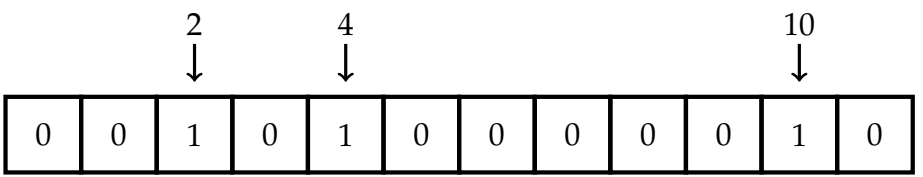
- união
- interseção
- diferença

Os multi-conjuntos são generalizações do conjunto, a multiplicidade é um elemento que nele ocorre.

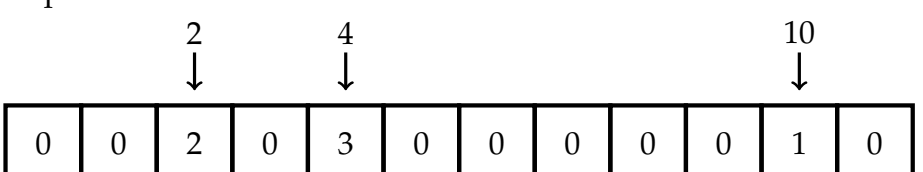
Conjuntos e Multi-conjuntos

O universo pode ser um segmento inicial do conjunto dos números naturais então a forma de implementar pode ser num array como os seguintes exemplos.

Exemplo: O conjunto $2, 4, 10$, é implementado num array (com pelo menos 11 posições) todas as componentes são 0, exceto as posições $2, 4, 10$ que devem ficar a 1.



Exemplo: O multi-conjunto $2, 2, 4, 4, 4, 10$, é implementado num array (com pelo menos 11 posições) todas as componentes são 0, exceto as posições $2, 4, 10$ que devem ficar a $2, 4$ e 1, respetivamente.



Conjuntos e Multi-conjuntos

A **inicialização** começa com um conjunto vazio, logo todas as posições do array é 0.

```
MAXS = 100

def initSet(s):
    s = [0 for i in range(MAXS)]
    return s
```

```
MAXMS = 100

def initMSet(s):
    s = [0 for i in range(MAXMS)]
    return s
```

Podemos também verificar se o array está vazio percorrendo todo o array.

```
def emptySet(s):
    r = 1
    i = 0
    while i < MAXS and r == 1:
        if s[i] != 0:
            r = 0
        i += 1
    return r
```

```
def emptyMSet(s):
    r = 1
    i = 0
    while i < MAXMS and r == 1:
        if s[i] != 0:
            r = 0
        i += 1
    return r
```

Conjuntos e Multi-conjuntos

O *custo* da inicialização e da verificação é compensado pela eficiência da adição de um novo elemento e da procura.

```
1 def addSet(s, x):
2     s[x] = 1
3     return 0
```

```
1 def addMSet(s, x):
2     s[x] += 1
3     return s[x]
```

```
1 def searchSet(s, x):
2     return s[x]
```

```
1 def searchMSet(s, x):
2     return s[x]
```

Para as outras operações é preciso percorrer o array.

Conjuntos e Multi-conjuntos

A união de 2 conjuntos corresponde ao menos conjunto que contem os 2, o mesmo acontece com os multi-conjuntos.
A interseção corresponde ao maior conjunto contido nos 2. E o mesmo se passa nos multi-conjuntos.

Exercício

Implemente em Python a união e a interseção de conjuntos e de multi-conjuntos.

Estas implementação são apenas possíveis quando o universo é um segmento inicial do conjunto dos números naturais. Este universo não pode ser muito grande. Nos casos em que tal não acontece, o normal é guardar informações sobre os elementos do conjunto.

Nota: em python existe a função `set()` que transforma tudo as listas em conjuntos. Nesta função os conjuntos são definidos de forma diferente.

SEQUÊNCIAS

Um sequência é um tipo de dados capaz de armazenar um número variável de itens.

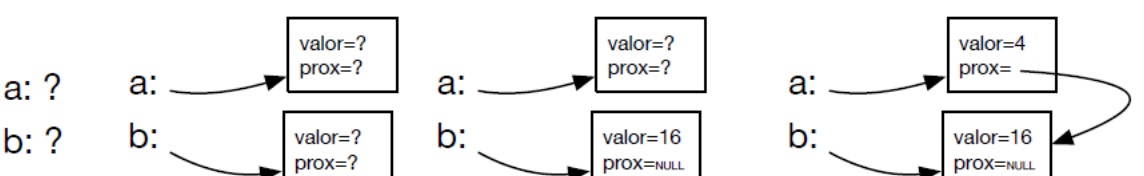
Tipicamente, as linguagens de programação como C usam a capacidade de alocar e libertar memória durante a execução do programa para definir uma sequência.
Em Python a alocação e libertação de memória são operações automáticas.

Para representar uma lista de inteiros podemos ter:

```
1 class Lista:
2     def __init__(self, valor=None, prox=None):
3         self.valor = valor
4         self.prox = prox
```

SEQUÊNCIAS

```
1 Criar duas estruturas Lista a e b
2 Atribuir 16 ao campo valor de b
3 Atribuir NULL ao campo prox de b
4 Atribuir 4 ao campo valor de a
5 Atribuir o endereco de b ao campo prox de a
```



SEQUÊNCIAS

Para acrescentar um elemento ao inicio de uma lista podemos usar a função cons:

```
def cons(self, x):
    if x is not None:
        new = Lista(x, self)
        return new
    else:
        return None
```

Com esta função é possível reescrever o exemplo anterior como:

```
Definir duas variaveis do tipo ponteiro para 'Lista', 'a' e 'b'
Chamar a funcao cons com  argumentos '16' e 'NULL' e atribui o resultado a 'b'
Chamar a funcao cons com os argumentos '4' e 'b' e atribui o resultado a 'a'
```

SEQUÊNCIAS

Para calcular o comprimento da lista são seguidos os endereços dos vários elementos da lista:
Acrescentar elementos no inicio de uma lista é bastante simples e executa em tempo constante.
Por outro lado, acrescentar um elemento ao fim da lista precisa de percorrer toda lista.
Neste caso, qual é o complexidade desta operação?

Exercício

Como deve implementar a função `snoc` que percorre toda lista e adiciona no fim de lista um elemento?

SEQUÊNCIA

Para inverter a ordem dos elementos de uma lista não vazia podemos começar por inverter a ordem dos elementos da cauda da lista seguido de colocar o primeiro elemento da lista originar no final.

```
def reverse(self):
    if self.prox is None:
        r = self
    else:
        r = pt = self.prox.reverse()
        while pt.prox is not None:
            pt = pt.prox
        pt.prox = self
        self.prox = None
    return r
```

Exercício

Mostre que a função `reverseL` acima tem uma complexidade quadrática em função do número de elementos da lista argumento. Para isso comece por apresentar uma relação de recorrência que traduza essa complexidade e resolva essa recorrência.

SEQUÊNCIA

No caso da inserção ordenada temos de isolar o caso particular em que essa inserção se faz no inicio da lista: pode ser porque a lista está vazia ou por que o elemento a inserir é menor do que o primeiro elemento da lista. Se este caso não ocorre devemos descobrir onde o novo elemento será inserido.

```
def insere(self, x):
    new = Lista(x)
    if not self.valor or self.valor > x:
        new.prox = self
        return new
    else:
        ant = self
        pt = self.prox
        while pt and pt.valor < x:
            ant = pt
            pt = pt.prox
        new.prox = pt
        ant.prox = new
    return self
```

Por outro lado, para algumas linguagens é possível evitar particularizar o caso da inserção no inicio da lista com a mesma estratégia usada na função snoc.

... quadro

SEQUÊNCIA

```
def cloneL(self):
    if not self.valor:
        return None
    else:
        r = pt = Lista(self.valor)
        l = self.prox
        while l:
            pt.prox = Lista(l.valor)
            pt = pt.prox
            l = l.prox
        pt.prox = None
    return r
```

Este é outro exemplo no qual podemos usar a técnica de *duplo apontador* para não precisarmos de isolar o caso particular do início da lista.

... quadro

Buffer

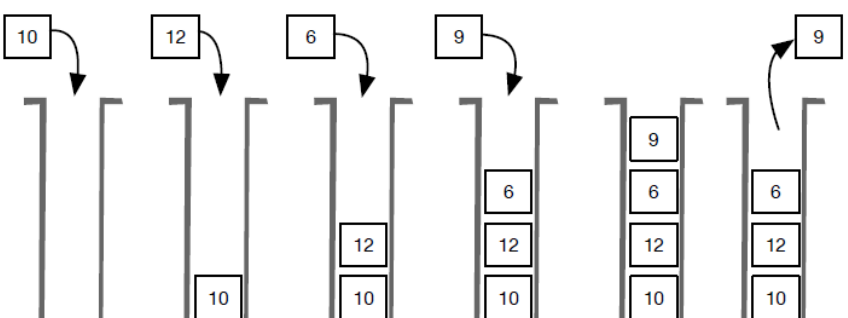
O buffer é um tipo de dados usados para armazenar itens e em que as operações fundamentais são a adição e a remoção de um elemento.
Distingue-se das sequências e dos conjuntos por não ter operações de procura e remoção de um item em particular, em vez disso, tem uma operação de remoção do próximo elemento.
A definição de qual é o próximo elemento é o que distingue as várias instanciações de buffer.

BUFFERS

STACKS

Na stack o primeiro elemento a ser removido é a último a ser inserido (*Last in First Out*).

O resultado de acrescentar (push) 10, 12, 6 e 9 é:



Ao removermos (pop) um elemento, removemos o 9.

BUFFERS

STACK

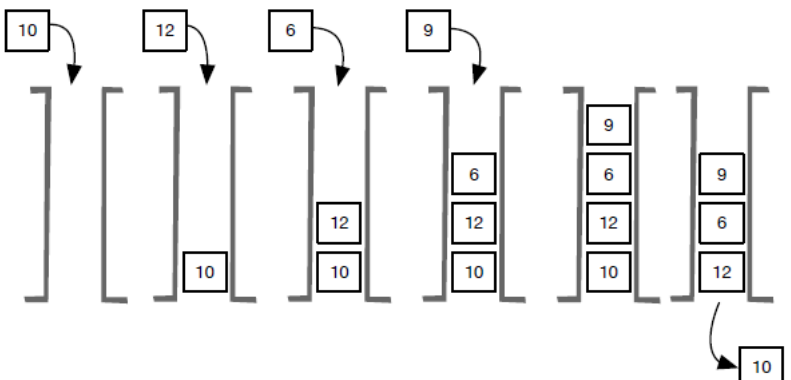
Implementações	Pros	Cons
Arrays estáticos	Fácil de implementar e acessar elementos em tempo constante	Tamanho fixo, não pode crescer ou diminuir dinamicamente
Listas ligadas	Pode crescer ou diminuir dinamicamente	Sobrecarga de memória extra para ponteiros, tempo de acesso mais lento
Arrays dinâmicos	Pode crescer ou diminuir dinamicamente, acesso constante aos elementos	Sobrecarga de memória extra para redimensionar o array

Exercício

Implemente a stack como um array estático. Tenha em atenção que este array estático deve ser implementado como uma classe com os métodos: `is_empty`, `push` e `pop`.

BUFFERS
QUEUES

O primeiro elemento a entrar na queue é o primeiro elemento a sair.
First in first out



Tal como a stack, uma queue pode ser implementada com arrays estáticos, listas ligadas e arrays dinâmicos.

Exercício

Implemente a queue como um array estático.

BUFFERS

FILAS COM PRIORIDADES

O elemento a ser retirado é o menor dos elementos armazenado.
Para implementar estes buffers temos duas alternativas:

- Manter os elementos do buffer num array ordenado (por ordem decrescente). Assim, a remoção de um elemento é bastante eficiente.
- Manter os elemtos do buffer num array armazanado por ordem de entrada no buffer. Assim, a inserção de um novo elemento é bastante eficiente.

Com estas alternativas se uma das operações (inserção e remoção) é eficiente a outra não é.

Uma alternativa melhor será usar uma heap ou min-heap.
A min-heap é uma árvore binária em que cada elemento é menor ou igual aos seus sucessores.

BUFFERS

FILAS COM PRIORIDADES

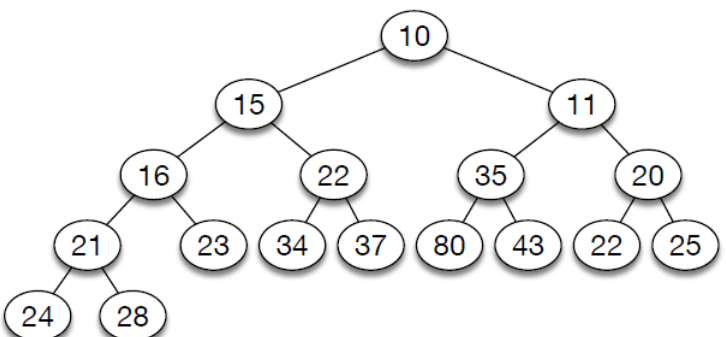
Para implementar a fila de prioridades as min.heaps devem ainda garantir que são árvores semi-completas (todas as folhas da árvore se encontram no último nível da árvore, ou à direita no nível anterior).
Assim podemos armazenar os elementos da árvore num array por níveis.
Para o elemento que está no nível `i`:

- o seu descendente esquerdo está no índice $2 * i + 1$
- o seu descendente direito está no índice $2 * i + 2$
- o seu ascendente está no nível $(i - 1)/2$

BUFFERS

FILAS COM PRIORIDADE (EXEMPLO)

Uma min-heap possível é:



Esta pode ser armazenada no array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
10	15	11	16	22	35	20	21	23	34	37	80	43	22	25	24	28

Como implementamos a remoção e a adição de elemento no array?

...quadro

Próximos episódios

- Dicionários
- Algoritmos sobre grafos.