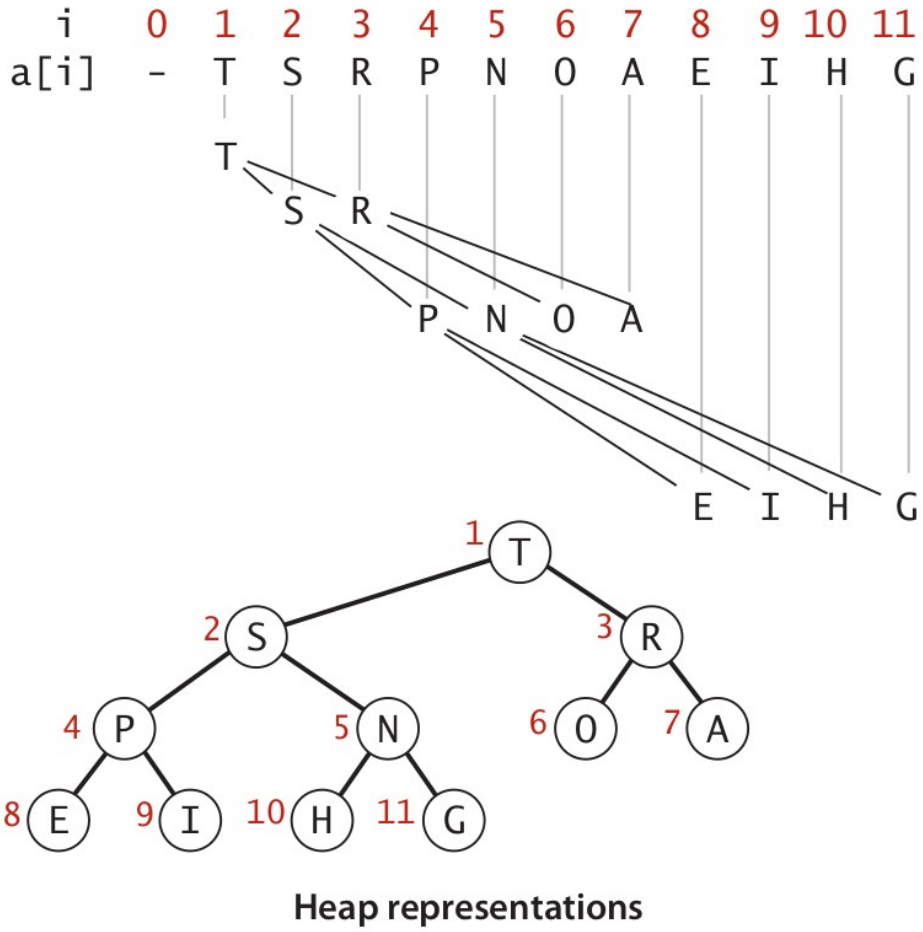
Fila de Prioridades

- É um **tipo abstrato de dados (TAD)** com chaves ordenadas
  - Representa valores
  - Acessadas por um conjunto de operações
    - Permite formar uma **interface** para uso de programas clientes
- Existem operações que **envolvem um grande volume** de informações que precisam de alguma ordenação
  - Não necessariamente precisam estar totalmente ordenados
    - O importante é saber qual tem a **maior prioridade**
  - Não necessariamente precisam processar todos os dados
    - Conforme novos dados forem coletados, atualiza-se a fila de prioridades
    - Podendo remover os com prioridades baixas para privilegiar os de maiores prioridades
  - Exemplos: transações financeiras, requisições de serviços, resultados de experimentos científicos, …
- A fila pode ser com prioridade máxima ou mínima, alterando a inserção e a remoção do máximo ou do mínimo
- **Operações**:
  - Remover o máximo valor de chave
    - Em caso de chaves duplicadas, máximo é qualquer chave com o maior valor
  - Inserir chaves

- **Interface (manipulação da fila)**:
  - PQinit(int maxN):
    - criar uma fila de prioridades com capacidade máxima inicial
  - PQinsert(Item v): inserir uma chave
  - PQdelmax(): retornar e remover a maior chave
  - PQempty(): testar se está vazia
- **Implementações**
  - Vetores e listas encadeadas:
    - Úteis para filas pequenas
    - Em vetor não ordenado:
      - Processo corresponde a executar um Selection Sort
      - Selecionar o de maior prioridade e colocar no ínicio
    - Em vetor ordenado
      - Processo corresponde a executar um Insertion Sort
      - A cada novo item, posicionar comparando com os seus antecessores
  - **Heap binária (heap):**
    - Estrutura de dados eficiente para as operações básicas da fila de prioridades
    - Ideia:
      - Organizar as chaves como em uma **árvore binária completa**:
        - Todos os níveis exceto o último estão cheios
        - Os nós do último nível estão o mais a esquerda possível

        - Cada nó possui **filhos com valores menores** ou iguais ao seu
        - **Raiz**: a **maior chave** da heap
        - Não ordena por completo, só garante-se que:
            - Quanto mais próximo à raiz, maior a prioridade
    - **Operações para manter a heap ordenada**:
        - Exige a navegação para cima e baixo
- **Implementações da árvore**:
    - Com listas encadeadas:
        - Necessário 3 ponteiros: para os filhos e pai
    - Vetor:
        - Representação sequencial da árvore
        - Níveis da árvore acessada pelos seus índices
            - Raiz: posição 1
            - Filhos: 2 e 3
            - Netos: 4, 5, 6 e 7
            - E assim por diante.
        - São estruturas mais rígidas, limitadas
        - Porém, para heaps, há flexibilidade suficente para a implementação das operações em tempo logarítmico
            - Remover o valor máximo (ou mínimo)
            - Inserir
        - Vantagem: **acesso direto** aos nós sem necessidade de atualização dos apontadores

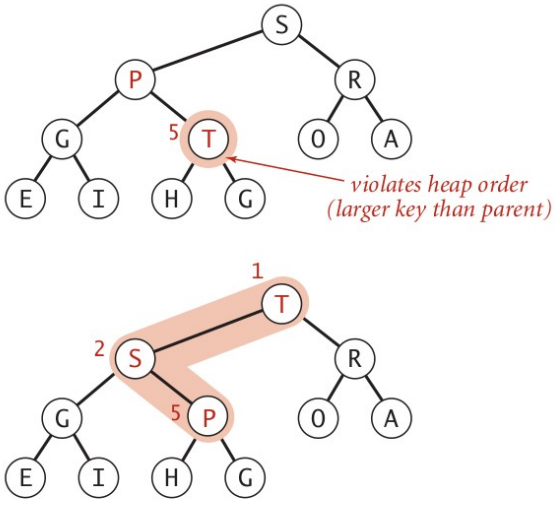
- **Heaps com vetores:**
  - Navegação trivial para cima e baixo:
    - Simples operação aritmética
    - Sendo um nó na posição k
      - pai: ⌊k /2⌋
      - filhos: 2k e 2k + 1



Heap representations

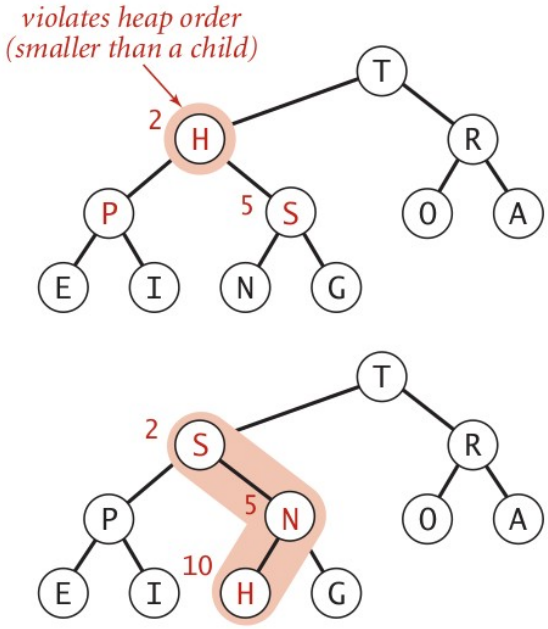
Detalhes de implementação

- Heap de **tamanho N** em um vetor **pq[]**
  - **pq[N+1]**: **pq[1..N]**
    - Não utiliza-se a posição pq[0] (??)
- **Operações**
  - **Prioridade aumentada ou chave inserida**
    - Inserção nas folhas da heap
    - **Restauração: subindo na heap**
  - **Prioridade diminuída ou chave removida**
    - Raiz da heap é substituída por uma folha
    - **Restauração: descendo na heap**
  - Inicialmente violam as condições da heap
    - pai com maior valor que seus filhos
  - Posteriormente, as chaves são reorganizadas para atender tais requisitos: restauração/conserto da heap
  - **Bottom-up (swim - fixUp):**
    - Sobe um nó com uma chave maior que seu pai
      - Troca de chave com seu pai
    - Recursivamente, sobe o nó até um pai maior ou a raiz
    - **Vamos implementar**



Bottom-up reheapify (swim)

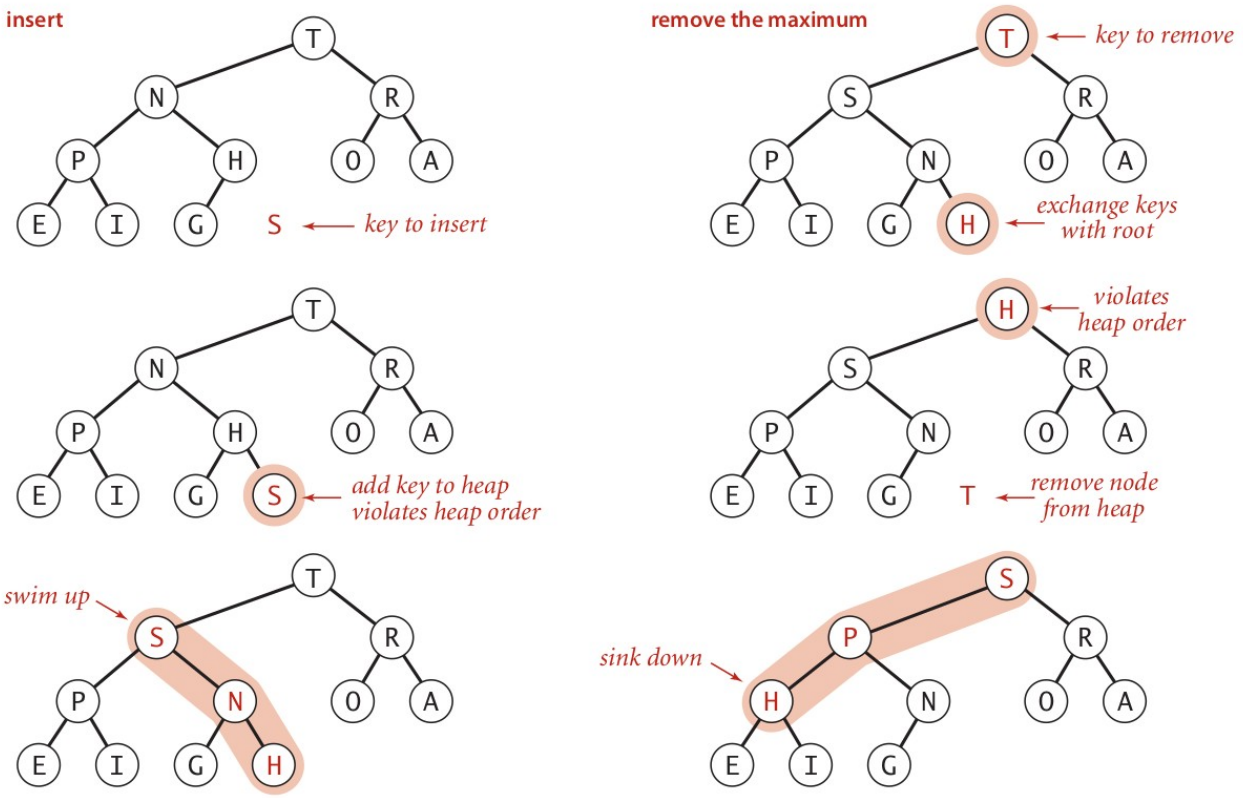
- **Top-down (sink - fixDown)**
  - Desce um nó com uma chave menor que um ou ambos os filhos
    - Troca de chave com o filho maior
  - Recursivamente, desce o nó até que ambos os filhos sejam menores (ou iguais) ou atingir a base
  - **Vamos implementar.**



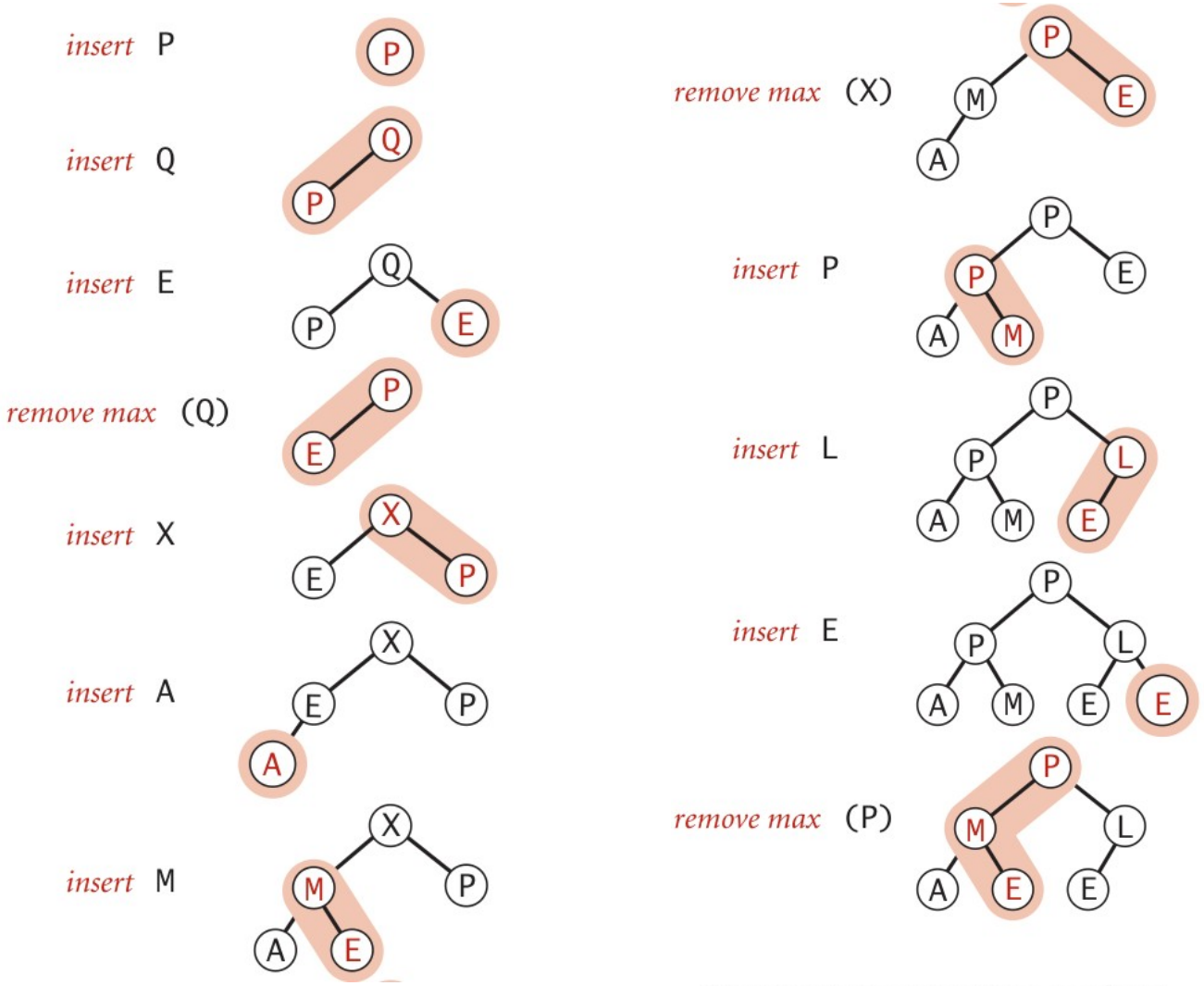
Top-down reheapify (sink)

- **Inserção**
  - Adicionar uma nova chave no fim do vetor
  - Incrementar o tamanho da heap
  - Restaura a ordenação da heap: "swim - **fixUp**"
  - Complexidade: 1+**logN** comparações - **O(logN)**

- **Remoção do máximo**
  - Remover a maior chave: **topo (raiz)**
  - **Coloca-se a chave do fim no topo**
  - Decrementa o tamanho da heap
  - Restaura a ordenação da heap: "sink - **fixDown**"
  - Complexidade: 2logN comparações

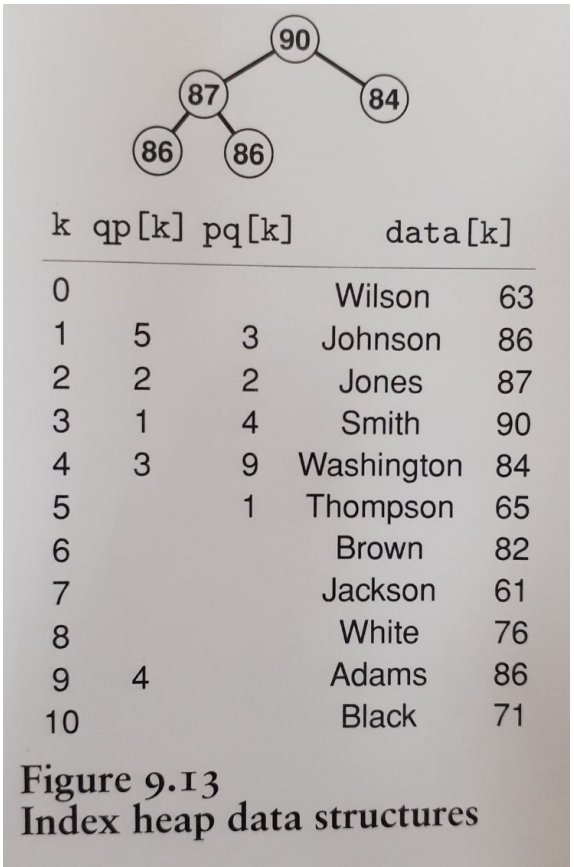


Heap operations



Priority queue operations in a heap

- Alterar o tamanho do vetor?
  - Duplicação do tamanho do vetor em insert() e redução em delMax(), assim como fizemos para pilhas
  - Os limites de tempo logarítmicos são amortizados quando o tamanho da fila de prioridade é arbitrário e os vetores são redimensionados
- E quando há **alteração no valor ou exclusão da chave**?
  - Se **temos o índice** na fila de prioridades é trivial
    - https://www.ic.unicamp.br/~rafael/cursos/2s2018/mc202/slides/unidade21-fila-de-prioridade.pdf
    - Cuidado com as buscas lineares
  - Se **não temos** acesso direto: fila de índices
- **Acesso pelo índice**
  - Fila de prioridades de vetor já existente
  - Permite a alteração dos valores das chaves e atualização "barata" da fila de prioridades
  - Fila de prioridades para índices
    - data[]: estrutura de dados com os dados
    - pq[k]:
      - fila de prioridades
      - pq[k]: índices de data[]
      - prioridade: comparação pelo conteúdo de data[]
    - qp[k]:

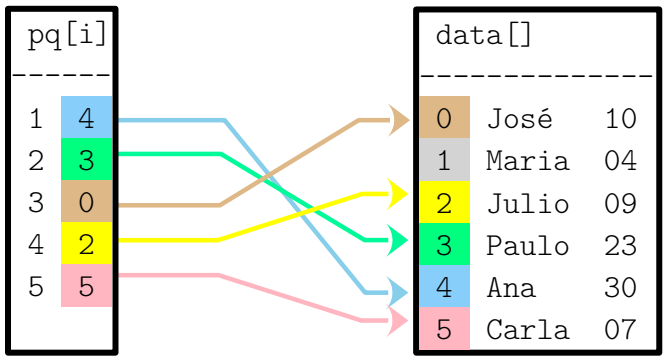


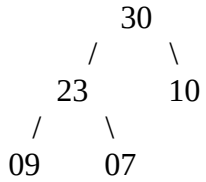
k	qp[k]	pq[k]	data[k]	
0			Wilson	63
1	5	3	Johnson	86
2	2	2	Jones	87
3	1	4	Smith	90
4	3	9	Washington	84
5		1	Thompson	65
6			Brown	82
7			Jackson	61
8			White	76
9	4		Adams	86
10			Black	71

Figure 9.13
Index heap data structures

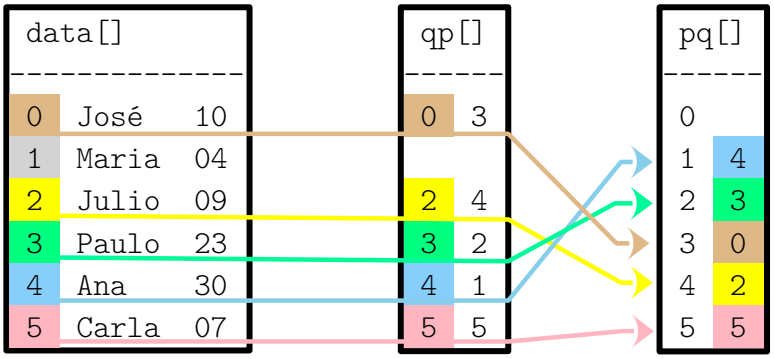
- lista de índices da fila
- k: índice de data
- acessar a fila pelo índice de data

- Da fila **pq[]**
  - Contém os índices de **data[]**
  - Acesso direto aos elementos de **data[]**

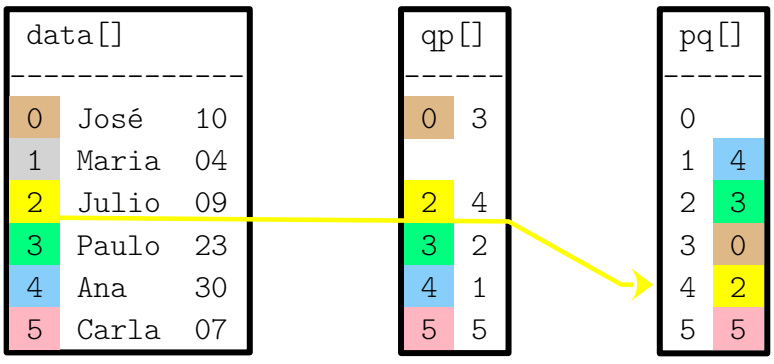


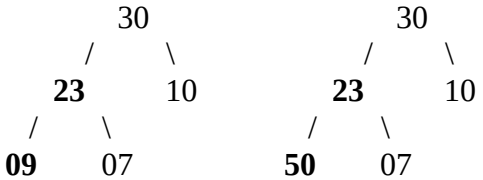


- De **data[]**
  - Contém os dados a serem organizados
  - Tem acesso direto à sua posição em **pq[]**??
  - E se houver alterações em **data[k]**?
    - Como atualizar a fila, se não sabemos a posição de **k** em **pq**?
      - **pq[i] = k**
    - Lista da localização de **k** em **pq**
      - **qp[k] = i ↔ pq[i] = k**
        - **k:** índice de **data[k]**
        - **i:** posição de **k** em **pq**

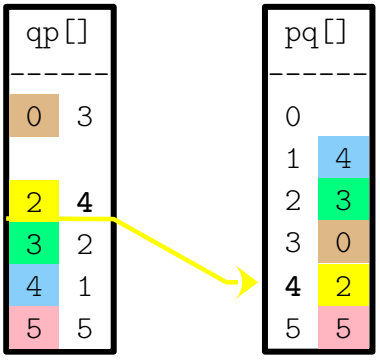


- Exemplo:
  - Prioridade de Julio mudou para 50
    - **data[2].nivel = 50**
    - Portanto, sua posição na fila deve ser alterada
    - Atualizar da fila de prioridades
      - **PQchange(2)**
      - Houve alteração em **data[2]**
        - **data[2] → qp[2] = 4**
        - Portanto, data[2] está posição 4 na fila:
          - **pq[4]**





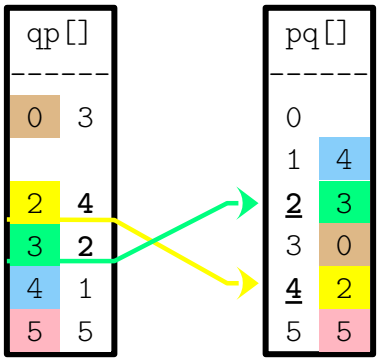
- Conserta a heap:
  - **fixUp(pq, qp[2])**
  - fixDown(pq, qp[2], N)



fixUp(pq, qp[2]) → fixUp(pq, 4)

exch(pq[k], pq[k/2]): troca com pai

exch(pq[4], pq[2]) → exch(2, 3)



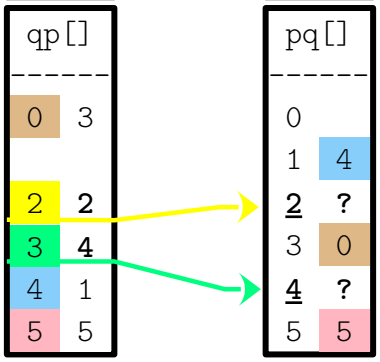
"i troca de prioridade com j"

"j troca de prioridade com i"

qp[2] = 4 → qp[2] = 2

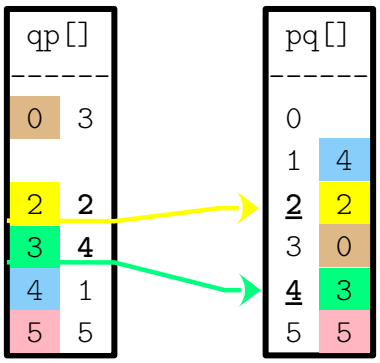
qp[3] = 2 → qp[3] = 4

```
int t = qp[2];
qp[2] = qp[4];
qp[4] = t;
```



"fila, coloca o i e o j nas novas posições"

```
qp[ i ] = nova_posicao_fila_de_i
qp[ j ] = nova_posicao_fila_de_j
pq[ nova_posicao_fila_de_i ] = i
pq[ nova_posicao_fila_de_j ] = j
```



pq[qp[2]] = 2 → pq[2] = 2

pq[qp[3]] = 3 → pq[4] = 3

- Interface de **fila de prioridades para índices**
  - **less(int i, int j)**
    - data[i].nivel < data[j].nivel
  - **exch(int i, int j)**
  - **PQchange(int k)**
  - PQinit()
  - PQempty()
  - PQinsert(int k)
    - Inserir o índice k de data[] em pq
  - PQdelmax()
  - **Vamos implementar.**

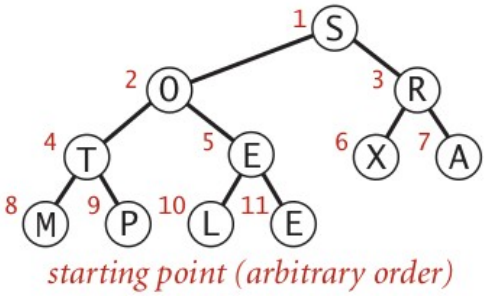
MÉTODO HEAP SORT

- **Construir** e **destrui** a heap da fila de prioridades
- Usar as **filas de prioridades** para ordenar elementos
  - **Fase 1: construção da heap (fila prioridades)**
    - Reorganiza o vetor em uma heap-ordenada
      - Topo é o de maior prioridade
      - Quanto mais próximo ao topo, maior a prioridade
        - Não há garantia de ordenação de todos os itens
  - **Fase 2: ordenação por remoção**
- **Solução 1:**
  - Usa-se somente a interface da TAD fila de prioridades
  - Criar uma fila de prioridades
    - Utiliza-se **espaço extra**
  - **Fase 1: construção da heap**
    - Construção da heap por inserção
      - Varredura da esquerda para direita
      - fixUp para posicionar na heap
      - Custo proporcional a **2NlogN**
  - **Fase 2: ordenação (decrescente)**
    - Ordenação por remoção (maior prioridade)
      - Reorganização da fila de prioridades
    - Cada item removido volta para o vetor original
  - **Vamos implementar.**

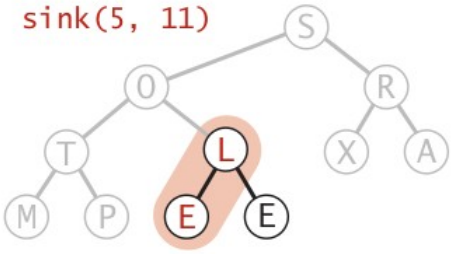
- **Solução 2 (otimização):**
  - Usa-se diretamente as funções exclusivas da TAD fila de prioridades: fixUp(swim), fixDown(sink)
  - Não há a necessidade de **espaços extras**
    - Vetor original é utilizado para construir a heap
  - **Fase 1: construção da heap**
    - Varredura da direita para esquerda
    - fixDown para preservar a heap-ordenada
    - Cada fixDown, constrói uma sub-heap
      - Cada posição no vetor é uma raiz de uma sub-heap
    - Percurso:
      - Inicializa da metade do vetor
        - **N/2**: pai dos nós folhas
        - Pular sub-heaps de tamanho 1
      - Termina na posição 1
    - Resultado (contra intuitivo):
      - Primeiro elemento sendo o maior elemento do vetor
      - Outros maiores elementos, próximos ao início
    - Custo proporcional a **2N**
  - **Fase 2: ordenação (decrescente)**
    - Remover o máximo repetidamente
      - Troca-se o último elemento pela raiz
      - Diminui-se o tamanho da fila
      - fixDown da raiz
  - **Vamos implementar.**

		a[i]											
N	k	0	1	2	3	4	5	6	7	8	9	10	11
initial values			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E

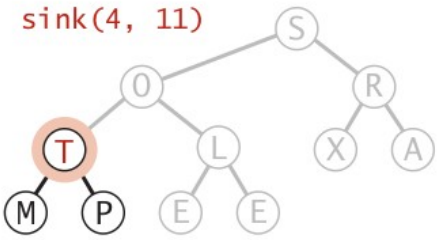
heap construction



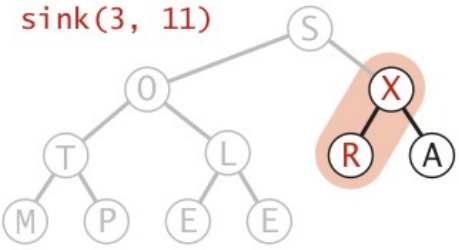
starting point (arbitrary order)



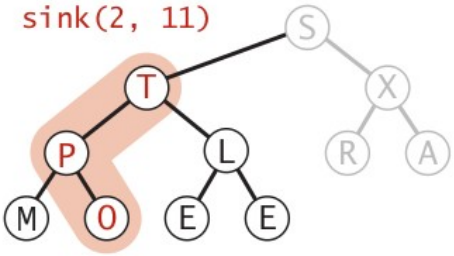
(1)



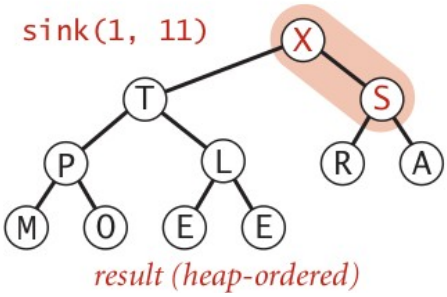
(2)



(3)



(4)

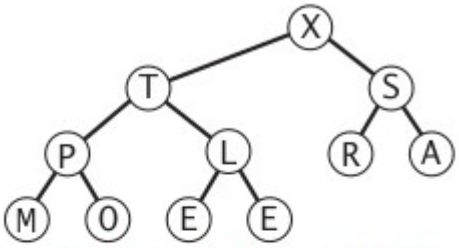


result (heap-ordered)
(5)

heap-ordered		X	T	S	P	L	R	A	M	O	E	E
10	1	T	P	S	O	L	R	A	M	E	E	X
9	1	S	P	R	O	L	E	A	M	E	T	X
8	1	R	P	E	O	L	E	A	M	S	T	X
7	1	P	O	E	M	L	E	A	R	S	T	X
6	1	O	M	E	A	L	E	P	R	S	T	X
5	1	M	L	E	A	E	O	P	R	S	T	X
4	1	L	E	E	A	M	O	P	R	S	T	X
3	1	E	A	E	L	M	O	P	R	S	T	X
2	1	E	A	E	L	M	O	P	R	S	T	X
1	1	A	E	E	L	M	O	P	R	S	T	X
sorted result		A	E	E	L	M	O	P	R	S	T	X

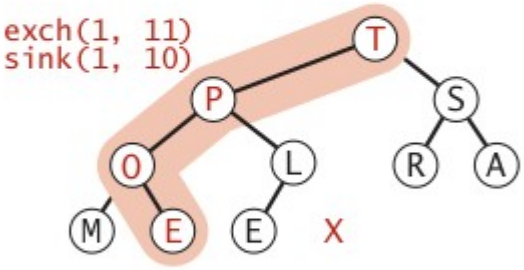
sortdown

(1)

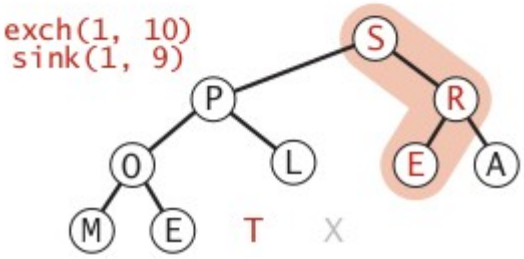


starting point (heap-ordered)

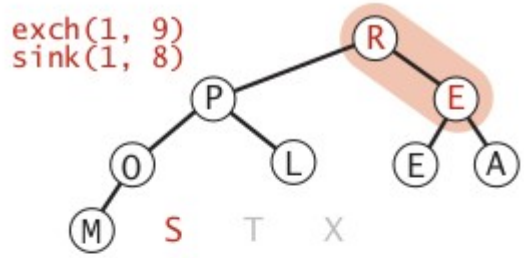
(2)



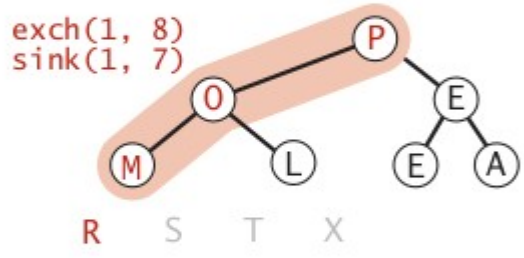
(3)



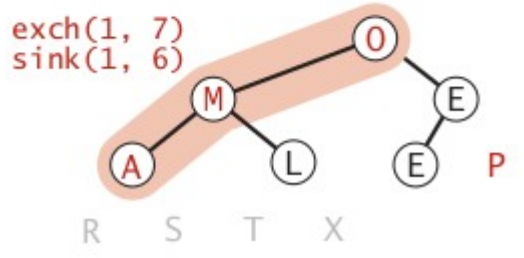
(4)



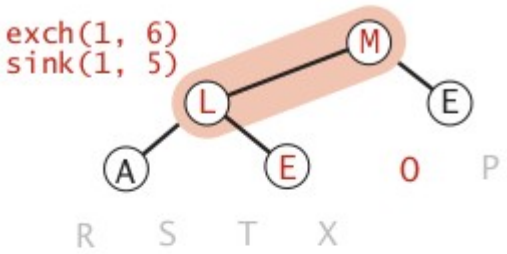
(6)



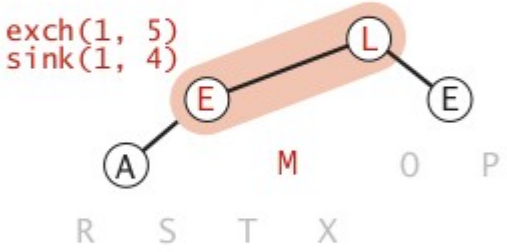
(7)



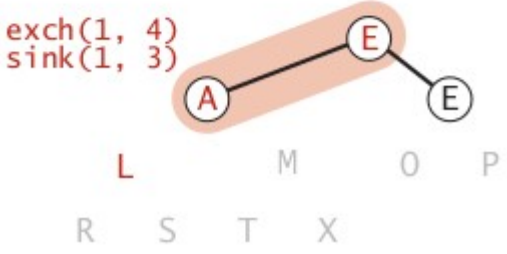
(8)



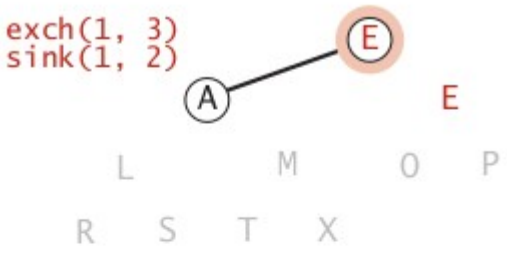
(9)



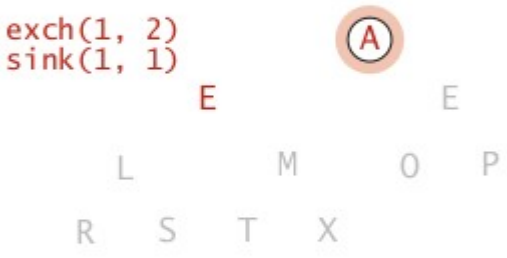
(10)



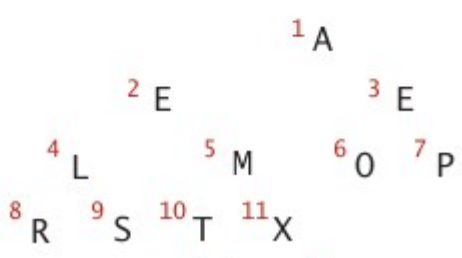
(11)



(12)



(13)



result (sorted)

- A **segunda fase** é a mais custosa
  - Reorganizar o heap a cada remoção
  - Porém a estrutura da heap (pseudo-ordenada) contribui na tarefa de encontrar o maior elemento

- **Complexidade**
  - Cerca de **2NlogN + 2N** comparações
    - **2N** na construção da heap
    - **2NlogN** no conserto da heap (segunda fase)

- **In-place?**
  - Utiliza memória extra significativa? Não.
  - In-place: sim.
- **Estabilidade?**
  - Mantém a ordem relativa?
  - Não é estável
- **Adaptatividade?**
  - Ordenação ajuda a melhorar o desempenho?
  - Não diminui as chamadas fixDown?
    - Há uma redução das chamadas iterativas na construção da heap (2N)
    - Porém, na ordenação final (segunda fase) não contribui no fixDown dos elementos folhas (2N log N)
    - Complexidade: O(nlogn)

algorithm	stable?	in place?	*order of growth to sort N items* running time	extra space	notes
selection sort	no	yes	N^2	1	
insertion sort	yes	yes	between N and N^2	1	depends on order of items
shellsort	no	yes	$N \log N$? $N^{6/5}$?	1	
quicksort	no	yes	$N \log N$	$\lg N$	probabilistic guarantee
3-way quicksort	no	yes	between N and $N \log N$	$\lg N$	probabilistic, also depends on distribution of input keys
mergesort	yes	no	$N \log N$	N	
heapsort	no	yes	$N \log N$	1	

Performance characteristics of sorting algorithms

MÉTODO INTRO SORT

- É uma importante combinação de algoritmos de ordenação interna, utilizado na biblioteca STL (Standart Template Libary) da linguagem C++

- Híbrido:
  - quick + merge + insertion
  - quick + heap + insertion

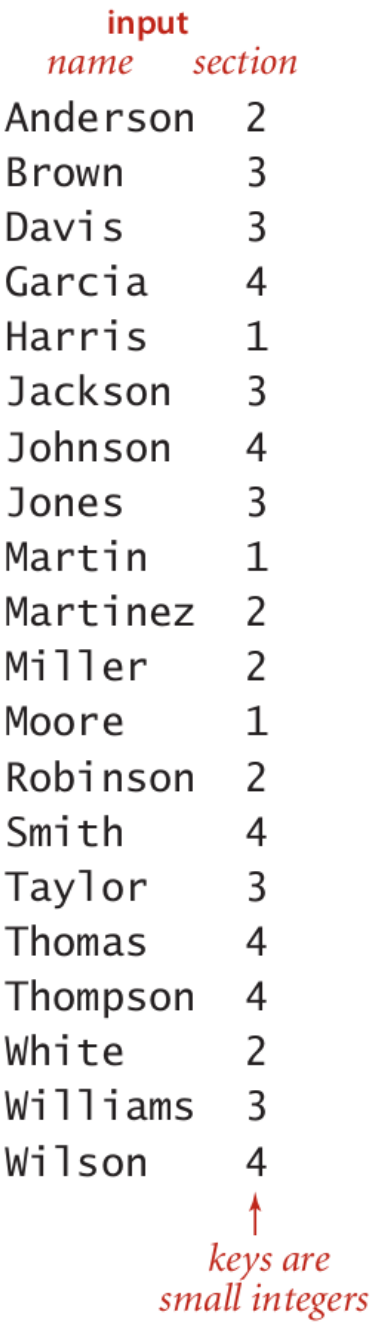
- Solução para utilizar as eficiências e evitar as deficiências de cada método
  - insertion: pequenos vetores, quase ordenados
  - quick: bom desempenho na maioria dos casos
    - quando a profundidade da recursividade atinge um máximo estipulado, aterna-se para outro método de ordenação

- **Complexidade** no pior caso: O(nlogn)
- **In-place?**
  - Merge:
    - Espaço extra: proporcional a N
    - Não
  - Heap
    - Sim
  - Quick

        - Espaço extra: proporcional a logN
        - Sim

- **Estabilidade?**
    - Merge
        - Estável
    - Quick e Heap
        - Não estável.
- **Adaptatividade?**
    - Não
- **Vamos implementar.**

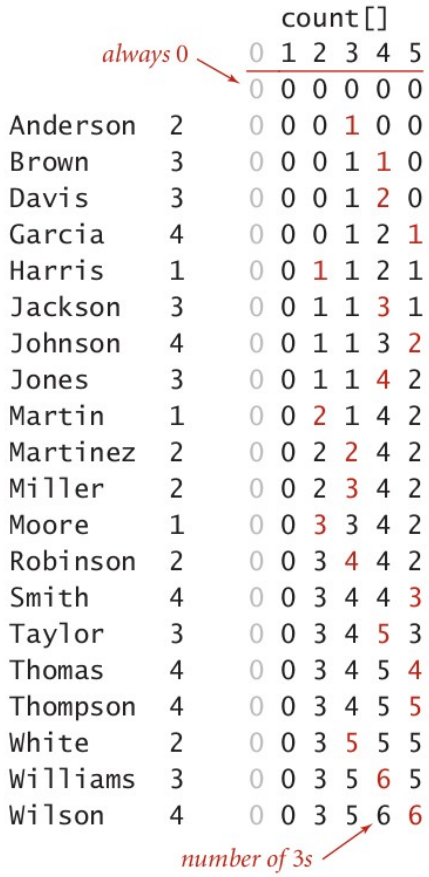
- **MÉTODO KEY-INDEXED COUNTING (COUNTING SORT)**
  - https://www.ic.unicamp.br/~rafael/cursos/2s2017/mc202/slides/unidade14-radixsort.pdf
  - Eficaz para chaves de números inteiros pequenos



  - Exemplo: alunos organizados por turmas numeradas 1, 2, 3 e assim por diante.
  - informações são mantidas em um vetor a[] de itens, cada um contendo um nome e um número da turma
    - a[i].key: número da turma do aluno indicado

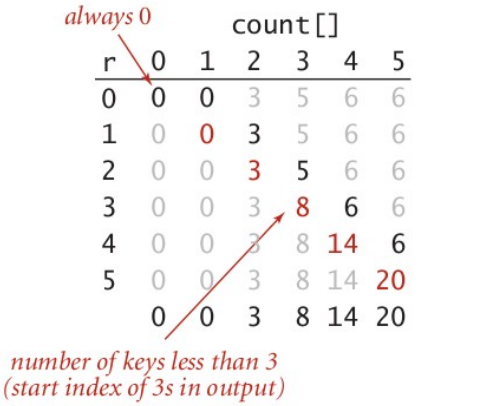
- **Contagem da frequência**
  - frequência de cada valor de chave, usando um array int count[]
  - cada posição: turma+1
    - count[3]=1
      - Anderson turma 2
    - count[4]=2
      - Brown e Davis turma 3
    - assim por diante
    - count[0] é sempre 0
    - count[i]
      - frequencia da chave i-1

```
for (i = 0; i < N; i++)
  count[a[i].key() + 1]++;
```



- **Transformar count em índice**
  - count[]: usado para definir as posições iniciais das chaves
  - chave 1:
    - count[1] posição da chave 1
    - 0 chaves 0 (count[1]) + 0 (count[0])
    - começa na posição 0
  - chave 2:

```
for (int r = 0; r < R; r++)
   count[r+1] += count[r];
```

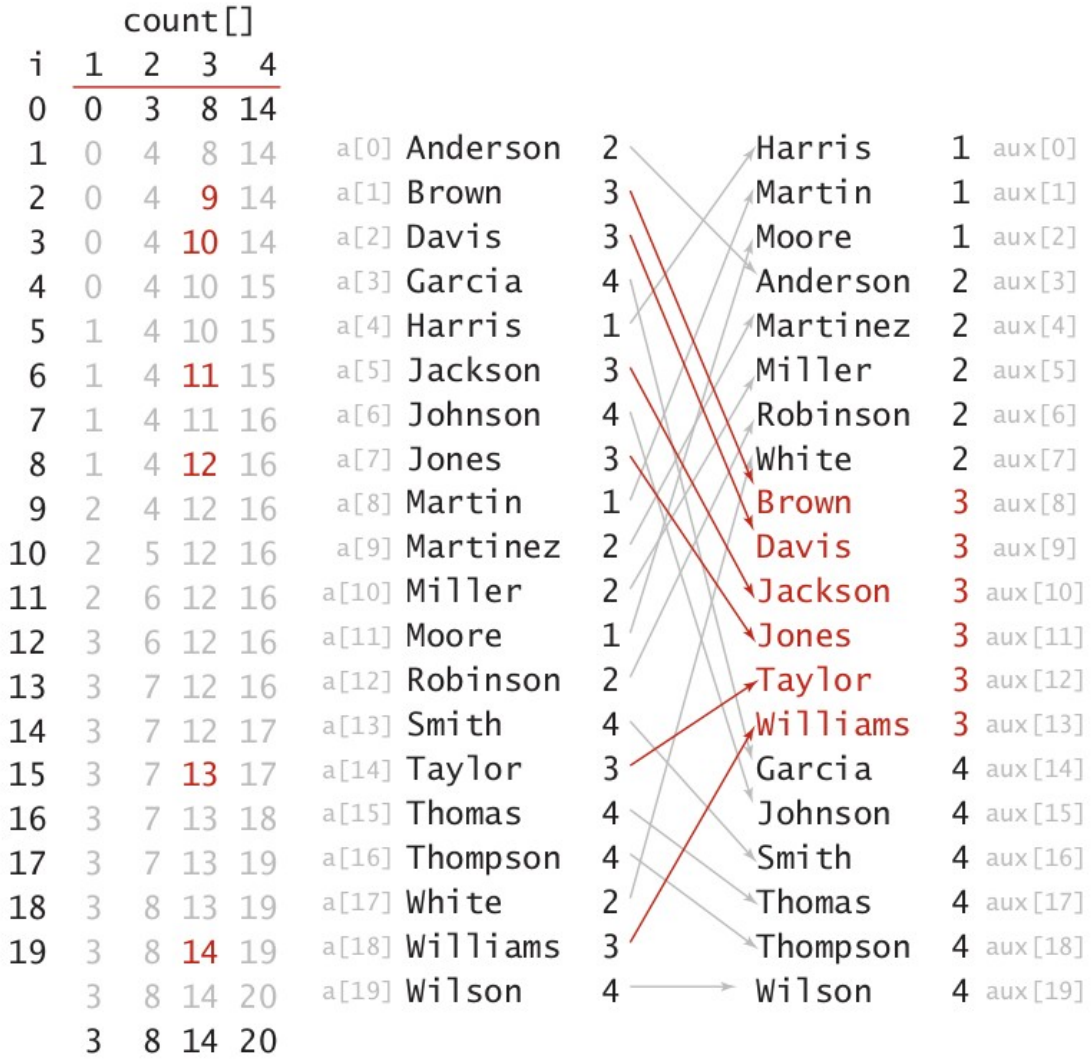


- chave[2] posição da chave 2
- 3 chaves 1 (count[2]) + 0 chaves 0 (count[1])
- começa na posição 3

- A posição é a soma da frequência de valores menores
  - Inicialmente count[i]
    - frequencia da chave i-1
    - posterior guarda a frequencia do anterior (por quê?)
  - Depois count[i]
    - posição da chave i

- **Distribuindo os dados**
  - count[] transformado em uma tabela de índice (posições)
  - ordena-se as chaves movendo os itens para um array auxiliar aux[]

```
for (int i = 0; i < N; i++)
   aux[count[a[i].key()]++] = a[i];
```



  - Estável: os itens com chaves iguais são reunidos, mas mantidos na mesma ordem relativa

- Última etapa: copiar aux para o vetor original

- Quando R (quantidade de chaves) é um fator constante de N (quantidade de elementos), temos uma classificação de tempo linear