# Filas de prioridade e heapsort

- Além do Sedgewick (**sempre** leiam o Sedgewick), veja
  - `http://www.ime.usp.br/~pf/algoritmos/aulas/hpsrt.html`

# Filas de prioridade e heapsort

- *Fila de prioridade*: fila na qual a operação de remoção atende um critério de *prioridade*
- Filas de prioridade podem ser usadas para ordenar itens: basta que a noção de prioridade seja compatível com a noção de ordem usada para a ordenação desejada [isto é, prioridade máxima = objeto de maior/menor chave]
- Com heaps, podemos implementar filas de prioridade na qual inserções e remoções podem ser feitas em tempo $O(\log N)$ (supondo que manipulamos no máximo $N$ objetos). Temos assim um algoritmo de ordenação com complexidade de tempo $O(N \log N)$: *heapsort*

# Filas de prioridade

```
/* prog9.1.c - PQ.h */
void PQinit(int);
 int PQempty();
void PQinsert(Item);
Item PQdelmax();
```

# Implementações

Alternativas:

- Vetor/lista; ordenado/não-ordenado
- Heaps
- Listas duplamente ligadas

# Implementação com “heaps”

- Itens armazenados em um vetor $v[1 \ldots N]$
- *Propriedade de heap*:

$$v[k] \leq v[\lfloor k/2 \rfloor] \tag{1}$$

para todo $1 < k \leq N$.

# Algoritmos sobre heaps

Ao aumentarmos a prioridade de `a[k]` (ou introduzirmos um elemento novo), podemos executar `fixUp(a, k)`:

```
/* prog9.3.c */
void fixUp(Item a[], int k)
  {
    while (k > 1 && less(a[k/2], a[k]))
      { exch(a[k], a[k/2]); k = k/2; }
  }
```

# Algoritmos sobre heaps

Ao diminuirmos a prioridade de `a[k]`, podemos executar `fixDown(a, k)`:

```
/* prog9.4.c */
void fixDown(Item a[], int k, int N)
  { int j;
    while (2*k <= N)
      { j = 2*k;
        if (j < N && less(a[j], a[j+1])) j++;
        if (!less(a[k], a[j])) break;
        exch(a[k], a[j]); k = j;
      }
  }
```

# Complexidade dos algoritmos para heaps

**Propriedade 1.** *Suponha que temos um heap com* $N$ *itens.*

- ▷ *A inserção de um* $(N+1)$*-ésimo elemento pode ser feita executando no máximo* $\log_2(N+1)$ *comparações entre itens.*
- ▷ *A operação de remoção de máximo pode ser feita executando no máximo* $2\lfloor \log_2(N-1) \rfloor$ *comparações entre itens.*

# Implementação de fila de prioridade com um heap

```
#include <stdlib.h>
#include "Item.h"

static Item *pq;
static int N;

void PQinit(int maxN)
  { pq = malloc((maxN+1)*sizeof(Item)); N = 0; }
 int PQempty()
  { return N == 0; }
```

# Implementação de fila de prioridade com um heap

```
void PQinsert(Item v)
  { pq[++N] = v; fixUp(pq, N); }
Item PQdelmax()
  {
    exch(pq[1], pq[N]);
    fixDown(pq, 1, N-1);
    return pq[N--];
  }
```

# Heapsort, versão preliminar

```
/* prog9.6.c */
void PQsort(Item a[], int l, int r)
  { int k;
    PQinit();
    for (k = l; k <= r; k++) PQinsert(a[k]);
    for (k = r; k >= l; k--) a[k] = PQdelmax();
  }
```

# Heapsort, versão preliminar

- A versão preliminar do heapsort tem complexidade de tempo $O(N \log N)$.
- Alguns melhoramentos (que não mudam a complexidade) são possíveis: não é necessário se fazer uma cópia dos dados em um heap; é possível se montar o heap de forma mais eficiente

# Heapsort, versão padrão

```
/* prog9.7.c */
#define pq(A) a[l-1+A]
void heapsort(Item a[], int l, int r)
  { int k, N = r-l+1;
    for (k = N/2; k >= 1; k--)
      fixDown(&pq(0), k, N);
    while (N > 1)
      { exch(pq(1), pq(N));
        fixDown(&pq(0), 1, --N); }
  }
```

# Construção de um heap

**Propriedade 2.** *A construção de um heap pode ser feita em tempo linear.*

- Chave: se $N = 2^n - 1$, então

$$\sum_{1 \le k < n} k 2^{n-k-1} = 2^n - n - 1 < N. \qquad (2)$$

  Para $N$ que não seja da forma $2^n - 1$, podemos provar a cota $2N$.

**Propriedade 3.** *Heapsort executa no máximo* $3\lfloor N \log_2 N \rfloor$ *comparações para ordenar* $N$ *itens.*

- Uma análise mais cuidadosa dá o limite $2N \log_2 N$.

# Seleção do $k$-ésimo elemento

**Exercício 4.** *Considere o problema da seleção do $k$-ésimo mínimo. Dê duas soluções para este problema:*

- *considerando a construção de um heap com os $N$ elementos seguida da remoção de $k$ itens (complexidade $O(N + k \log N)$);*
- *considerando a construção de um heap com $k$ elementos (complexidade $O(k + N \log k)$).*

# Filas de prioridade como uma ADT de primeira classe

```
/* prog9.8.c - PQ1st.h */
typedef struct pq* PQ;
typedef struct PQnode* PQlink;
    PQ PQinit();
   int PQempty(PQ);
PQlink PQinsert(PQ, Item);
  Item PQdelmax(PQ);
  void PQchange(PQ, PQlink, Item);
  void PQdelete(PQ, PQlink);
  void PQjoin(PQ, PQ);
```

# Implementação, com listas duplamente ligadas com cabeça e cauda

```
/* prog9.9.c */
#include <stdlib.h>
#include "Item.h"
#include "PQ1st.h"

struct PQnode { Item key; PQlink prev, next; };
struct pq { PQlink head, tail; };
```

# Implementação, com listas duplamente ligadas

```
/* prog9.9.c */
[...]

PQ PQinit()
  { PQ pq = malloc(sizeof *pq);
    PQlink h = malloc(sizeof *h),
           t = malloc(sizeof *t);
    h->prev = t; h->next = t;
    t->prev = h; t->next = h;
    pq->head = h; pq->tail = t;
    return pq;
  }
```

# Implementação, com listas duplamente

```
/* prog9.9.c */
[...]

int PQempty(PQ pq)
  { return pq->head->next->next == pq->head; }

PQlink PQinsert(PQ pq, Item v)
  { PQlink t = malloc(sizeof *t);
    t->key = v;
    t->next = pq->head->next; t->next->prev = t;
    t->prev = pq->head; pq->head->next = t;
  }
```

# Implementação, com listas duplamente ligadas

```
/* prog9.9.c */
[...]

Item PQdelmax(PQ pq)
  { Item max; struct PQnode *t, *x = pq->head->next;
    for (t = x; t->next != pq->head; t = t->next)
      if (t->key > x->key) x = t;
    max = x->key;
    x->next->prev = x->prev;
    x->prev->next = x->next;
    free(x); return max;
  }
```

# Implementação, com listas duplamente ligadas

```
/* prog9.10.c */
[...]

void PQchange(PQ pq, PQlink x, Item v)
  { x->key = v; }
void PQdelete(PQ pq, PQlink x)
  { PQlink t;
    t->next->prev = t->prev;
    t->prev->next = t->next;
    free(t);
  }
```

# Implementação, com listas duplamente ligadas

```
/* prog9.10.c */
[...]

void PQjoin(PQ a, PQ b)
  { PQlink atail, bhead;
    a->tail->prev->next = b->head->next;
    b->head->next->prev = a->tail->prev;
    a->head->prev = b->tail;
    b->tail->next = a->head;
    free(a->tail); free(b->head);
  }
```

# Filas de prioridade para índices

- Dados já na memória, por exemplo, como um vetor de itens
- Queremos manter uma fila para eles (ou parte deles)
- Vejamos uma implementação com heaps

# Uma fila de prioridade para índices, com heap

```
/* prog9.11.c - PQindex.h */
 int less(int, int);
void PQinit();
 int PQempty();
void PQinsert(int);
 int PQdelmax();
void PQchange(int);
void PQdelete(int);
```

# Uma fila de prioridade para índices, com heap

```
/* prog9.12.c */
#include "PQindex.h"
typedef int Item;
static int N, pq[maxPQ+1], qp[maxPQ+1];
void exch(int i, int j)
  { int t;
    t = i; i = j; j = t;
    t = qp[i]; qp[i] = qp[j]; qp[j] = t;
  }
void PQinit() { N = 0; }
 int PQempty() { return !N; }
```

# Uma fila de prioridade para índices, com heap

```
/* prog9.12.c */
[...]

void PQinsert(int k)
  { qp[k] = ++N; pq[N] = k; fixUp(pq, N); }
 int PQdelmax()
  {
    exch(pq[1], pq[N]);
    fixDown(pq, 1, --N);
    return pq[N+1];
  }
void PQchange(int k)
  { fixUp(pq, qp[k]); fixDown(pq, qp[k], N); }
```