COS 226 Lecture 5: Priority Queues

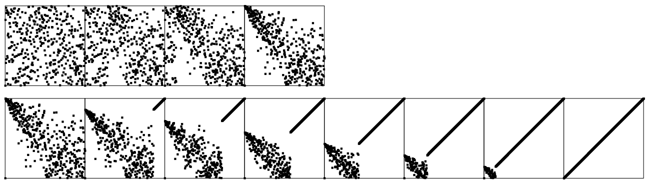
Abstract data types

- client
- interface
- implementation

Priority queue ADT

- insert
- remove the largest

HEAPS and Heapsort



BINOMIAL QUEUES

Abstract data types

Separate INTERFACE and IMPLEMENTATION

- easier maintainence of large programs
- build layers of abstraction
- reuse software
- elementary example: pushdown stack

INTERFACE: description of data type, basic operations
CLIENT: program using operations defined in interface
IMPLEMENTATION: actual code implementing operations

Client can't know details of implementation
(many implementations to choose from)
Implementation can't know details of client needs
(many clients use the same implementation)

Modern programming languages support ADTs

- C++, Modula-3, Oberon, Java (C)

ADTs and algorithms

PERFORMANCE MATTERS
ADT allows us to substitute better algorithms
without changing any client code

Running time depends on

- implementation
- client usage

Might need different implementations for different clients

GOALS

- general-purpose ADT useful for many clients
- efficient implementation of all ADT functions

ADTs provide levels of abstraction allowing us to build
algorithms for increasingly complicated problems
Ex: linked list -> stack -> quicksort

Priority queue ADT

Records with keys (priorities)

Two basic operations
INSERT
DELETE LARGEST
generic operations common to many ADTs

- create
- test if empty
- destroy (often ignored if not harmful)

Example applications

- simulation
- numerical computation
- compression algorithms
- graph-searching algorithms

Priority queue interface

INTERFACE for basic operations

```
.     void PQinit();
.     void PQinsert(Item);
.     Item PQdelmax();
.      int PQempty();
```

Should also specify
constraints and error conditions

Other useful operations

- delete a specified item
- change an item's priority
- merge together two PQs
- (stay tuned)

Sample PQ client

Find the M SMALLEST of N items (typical vals: M=100, N=1000000)

```
PQinit();
for (k = 0; k < M; k++) PQinsert(nextItem());
for (k = M; k < N; k++)
  {
    PQinsert(nextItem());
    t = PQdelmax();
  }
for (k = 0; k < M; k++) a[k] = PQdelmax(());
```

Time bounds for standard implementations:

- space proportional to M
- brute-force: N M
- best: N lg M
- best offline: N (with select, see lecture 3)

Unordered-array PQ implementation

```
static Item *pq;
static int N;
PQinsert(Item v)
  { pq[N++] = v; }
Item PQdelmax()
  {
    int j, max = 0;
    for (j = 1; j < N; j++)
      if (less(pq[max], pq[j])) max = j;
    exch(pq[max], pq[N]);
    return pq[--N];
  }
void PQinit(int maxN)
  { pq = malloc(maxN*sizeof(Item)); N = 0; }
int PQempty()
  { return N == 0; }
```


Other PQ implementations

Elementary

- ordered array
- unordered linked list
- ordered linked list

Advanced

- heap
- binomial queue

Client/Interface/Implementation

INTERFACE
- define data types
- declare functions
- in C, use ".h" file (no executable code)

CLIENT:
- include ".h" file
- call functions

IMPLEMENTATION:
- include ".h" file
- give code for functions

Client and implementation can be compiled
- at different times, then function calls
- LINKED to their implementations

Details: Sedgewick, Chapter 4; COS 217

Modular programming

Priority queue ADT (continued)

Other useful operations
- construct a PQ from N items
- return the value of the largest
- delete a specified item
- change an item's priority
- merge together two PQs

Interface more complicated
- need HANDLES for records
- need HANDLES for priority queues
- where's the data?
  (client, implementation, or both?)

First-class PQ ADT

```
.  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);
.       PQ PQjoin(PQ, PQ);
```

PQ and PQlink are pointers to structures
- to be specified in the implementation

More info: section 4.8 in Sedgewick; lecture 7

PQ implementations cost summary

Worst-case per-operation time as a function of PQ size

	insert	delete max	delete	find max	change key	join
ordered						
array	N	1	N	1	N	N
list	N	1	1	1	N	N
unordered						
array	1	N	1	N	1	N
list	1	N	1	N	1	1
heap	lg N	lg N	lg N	1	lg N	N
binomial queue	lg N	lg N	lg N	lg N	lg N	lg N
best in theory	1	lg N	lg N	1	1	1

PQ data structures

HEAP
- lg N for all operations

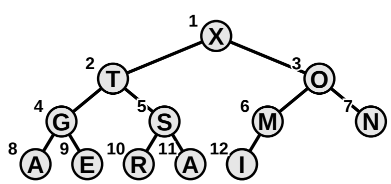
BINOMIAL QUEUE
- lg N for all operations
- constant (amortized) for most
- basis for near-optimal slgs

Algorithm design success story:
- nearly optimal worst-case cost
- simple (but ingenious!) algorithms
- costs even lower in practice

Heap-ordered complete binary trees

COMPLETE BINARY TREE:
- leaves on two levels, on left at bottom level

HEAP-ORDERED:
- parent larger than both children
- therefore, largest at root
- can define for any tree, not just complete



Heap

Array representation of heap-ordered binary tree

- root in a[1]
- children of 1 in a[2] and a[3]
- children of i in a[2i] and a[2i+1]
- parent of i in a[i/2]

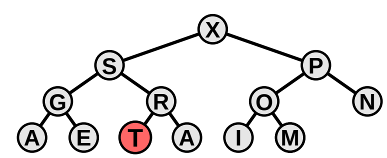
No explicit links needed for tree

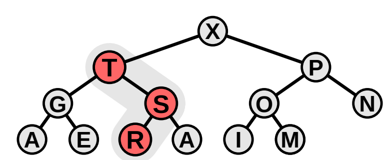
.	0	1	2	3	4	5	6	7	8	9	10	11	12
.		X	T	O	G	S	M	N	A	E	R	A	I

Promotion (bubbling up in a heap)

Change key in node at the bottom of the heap

To restore heap condition:
- exchange with parent if necessary





Promotion implementation

Peter principle

- nodes rise to level of incompetence

Node k's parent in heap is k/2

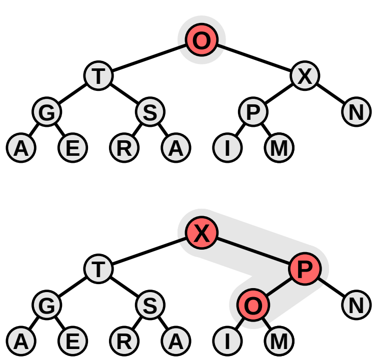
```
fixUp(Item a[], int k)
  {
    while (k > 1 && less(a[k/2], a[k]))
      { exch(a[k], a[k/2]); k = k/2; }
  }
```


Demotion (sifting down in a heap)

Change key in node at the top of the heap

To restore heap condition:

- exchange with larger child if necessary



Demotion implementation

"Power struggle" principle

- better subordinate is promoted

Node k's children in heap are 2k and 2k+1

```
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;
      }
  }
```

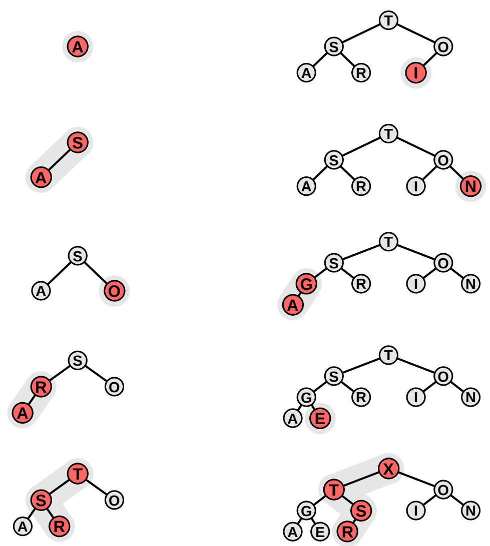

PQ implementation with heaps

PQinsert: add node at bottom, bubble up

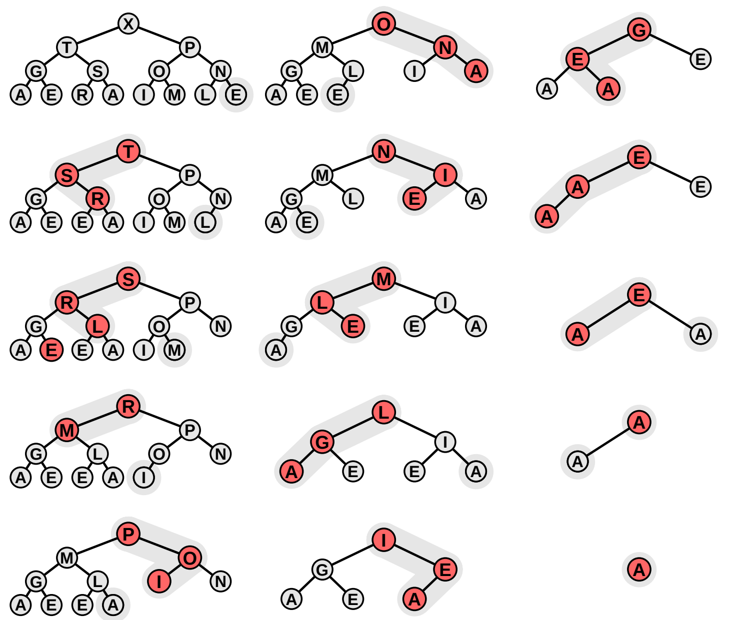
PQdelmax: exch root with node at bottom, sift down

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

Constructing a heap (top-down)



Sorting down a heap



Heapsort

Abandon ADT concept to save space

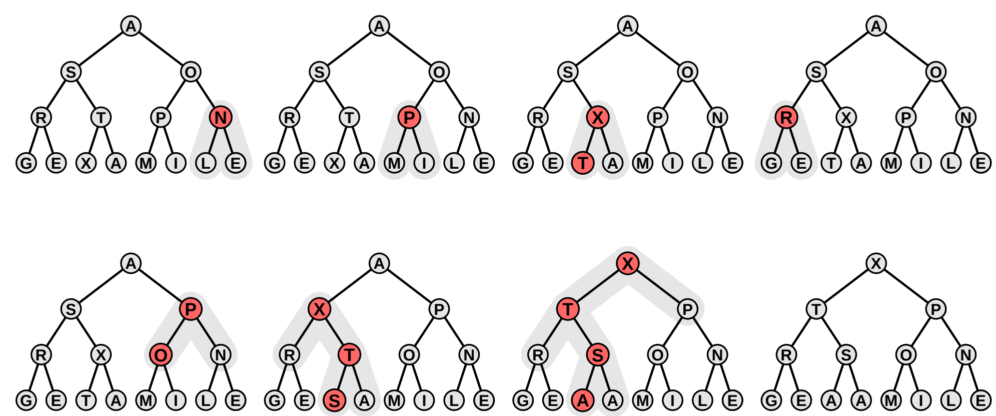
Faster to construct heap backwards

```
#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);
      }
  }
```

Widely used sorting method

- inplace, guaranteed NlgN time

Bottom-up heap construction



Binomial queues

Support ALL PQ operations in lgN steps

- Heaps have slow merge

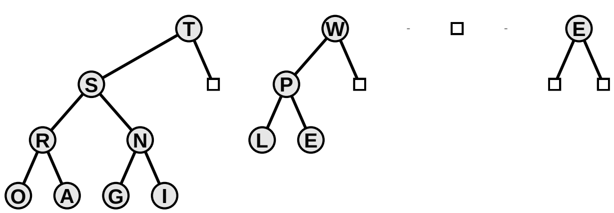
Def: In a LEFT HEAP-ORDERED tree, each node
is larger than all nodes in left subtree

Def: A POWER-OF-2 TREE is a binary tree

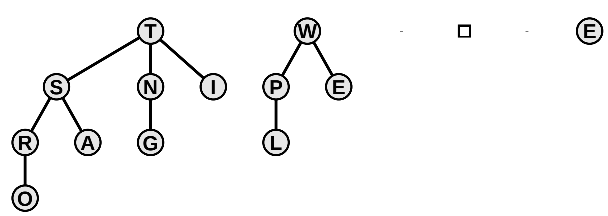
- left subtree of root complete
- right subtree empty
- (therefore, 2^n nodes)

Def: A BINOMIAL QUEUE of size N is
of left heap-ordered power-of-2 trees
one for each 1 bit in binary rep. of N

Binomial queue example



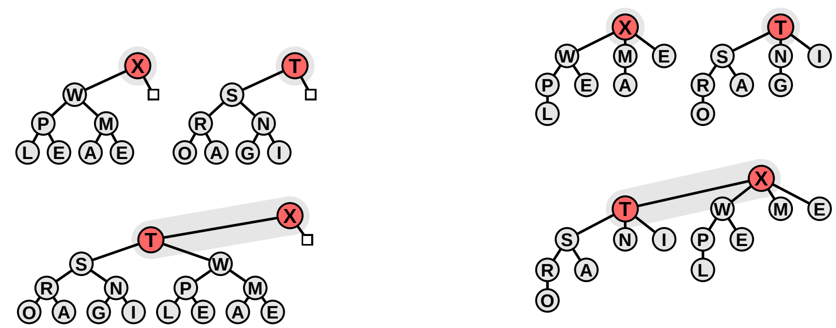
Corresponds to heap-ordered forest:



Joining power-of-2 heaps

Constant-time operation

- larger of two roots at top
- left subtree to right subtree of other root
- result is left-heap-ordered if inputs are



Joining power-of-2 heaps (code)

Representation

- two pointers per node
- need HANDLE (pointer to node)

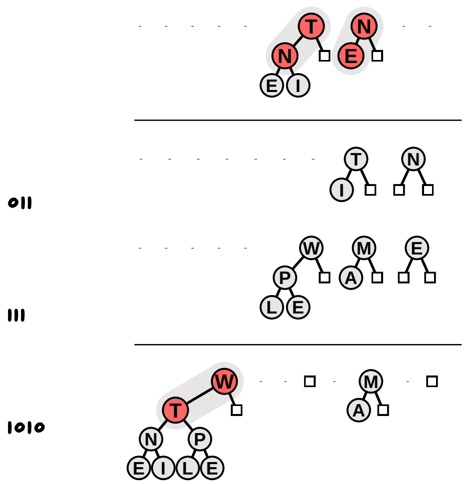
```
struct PQnode
  { Item key; PQlink l, r; };
struct pq { PQlink *bq; };
```

```
PQlink pair(PQlink p, PQlink q)
  { PQlink t;
    if (less(p->key, q->key))
      { p->r = q->l; q->l = p; return q; }
    else
      { q->r = p->l; p->l = q; return p; }
  }
```

Joining binomial queues

Corresponds to adding binary numbers

- 1 bits correspond to power-of-2 heaps
- 1+1=10 corresponds to carry



Joining binomial queues (carry table)

c	b	a	a	c
0	0	0	a	0
0	0	1	a	0
0	1	0	b	0
0	1	1	0	a+b
1	0	0	c	0
1	0	1	0	a+c
1	1	0	0	b+c
1	1	1	a	b+c

Joining binomial queues (code)

```
#define test(C, B, A) 4*(C) + 2*(B) + 1*(A)
void PQjoin(PQlink *a, PQlink *b)
{ int i; PQlink c = z;
   for (i = 0; i < maxBQsize; i++)
      switch(test(c != z, b[i] != z, a[i] != z))
      {
      case 2: a[i] = b[i]; break;
      case 3: c = pair(a[i], b[i]);
                  a[i] = z; break;
      case 4: a[i] = c; c = z; break;
      case 5: c = pair(c, a[i]);
                  a[i] = z; break;
      case 6:
      case 7: c = pair(c, b[i]); break;
      }
}
```


Binomial queues summary

BQ of size N is array of power-of-two heaps

- one for each bit in binary rep. of N

Joining two BQs is like adding binary numbers

- insert is like incrementing
- delete, delmax are like decrementing
- heap-like promotion, demotion for "change priority"

Guaranteed performance: lgN per operation

Amortized performance: constant per operation

Ex: PQinsert N items, then one more

- N even, just insert item
- N = ...01, just two steps
- N = ..011, just three steps
- total cost LINEAR: N/2 + 2(N/4) + 3(N/8) + 4(N/16) + ...

Basis for advanced data structures

Good candidate for library PQ implementation