COS 226 Lecture 5: Priority Queues

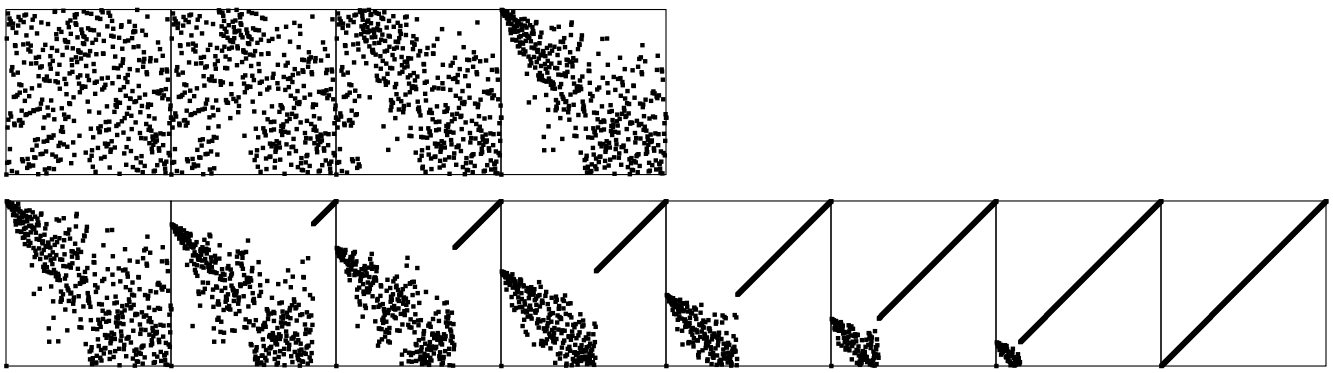
Abstract data types
- client
- interface
- implementation

Priority queue ADT
- insert
- remove the largest

Brute-force implementations

HEAPS

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)

Caveats

- perspective needed
- performance matters

Priority queue ADT

Records with keys (priorities)

Two basic operations

* insert
* delete the largest

generic operations common to many ADTs

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

INTERFACE for basic operations

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

Should also specify

constraints and error conditions

Example applications

* simulation
* numerical computation
* compression algorithms
* graph searching algorithms

Sample client program

Find the M SMALLEST of N items

Typical values: 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(());
```

Cost? Depends on PQ implementation

Reasonable assumption for any implementation:

space proportional M

Time bounds for standard implementations:

brute-force: N M

best: N lg M

Typical brute-force implementation

Use an unordered array

```
static itemType pq[];
static int N;

PQinsert(itemType v)
   { pq[++N] = v; }
itemType PQdelmax()
   {
      int j, max = 1;
      for (j = 1; j <= N; j++)
         if (pq[j] > pq[max]) max = j;
      exch(pq[max], pq[N]);
      return pq[N--];
   }
PQinit()
   { N = 0; }
int PQempty()
   { return !N; }
```

Should also test and report errors

Other options for brute-force implementations

* ordered array
* unordered singly-linked list
* ordered singly-linked list

Client/Interface/Implementation

Separate compilation
(Freudian slip: "complication")

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: COS 217

Modular programming
separate compilation abstraction
essential in implementing big systems
useful in putting good algs to work

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)

PQ implementations cost summary

Worst-case per-operation time cost
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**	**lg N**	**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**

Algorithm design success story:

* nearly optimal worst-case cost
* simple (but ingenious!) algorithms

Costs lower in practice

average-case analysis

amortized analysis

Heap-ordered complete binary trees

COMPLETE BINARY TREE:

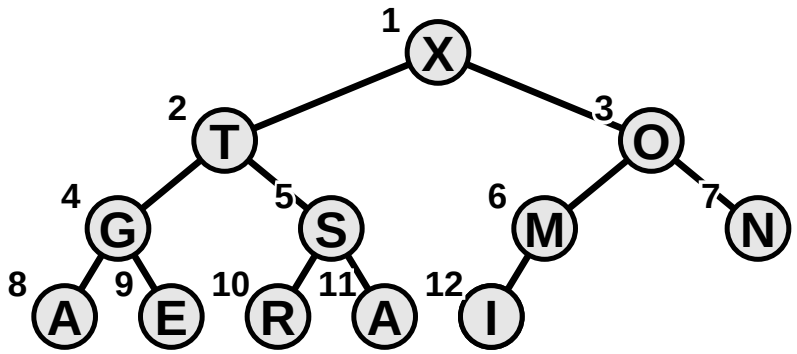
leaves on two levels, on left at bottom level

HEAP-ORDERED:

parent larger than both children

therefore, largest at root

can define for any tree, not just complete



Array representation:

root in a[1]

children of root 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

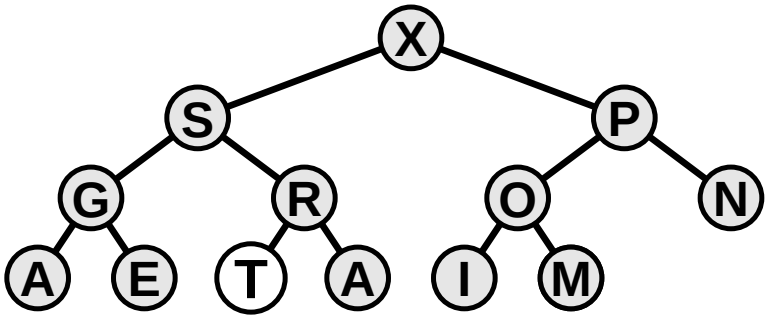
.	0	1	2	3	4	5	6	7	8	9	10	11	12
.		X	T	O	G	S	M	N	A	E	R	A	I

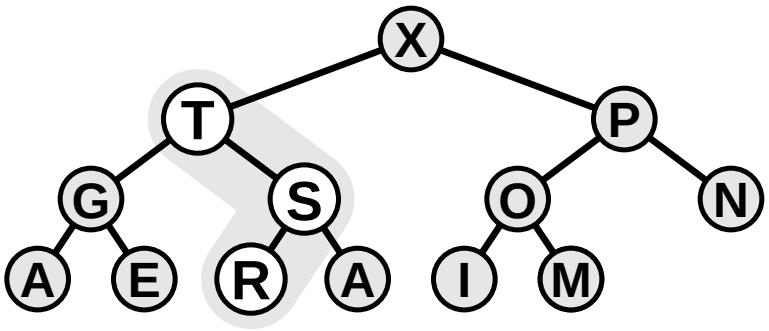
Promotion (bubbling up in a heap)

Change key in node at the bottom of the heap

To restore heap condition:

exchange with parent if necessary





"How to Succeed" model (Peter Principle)

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

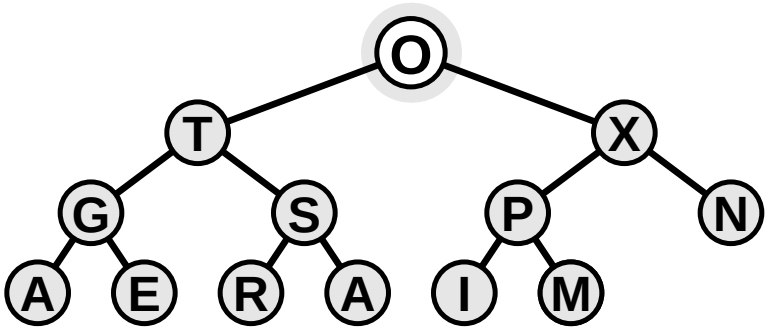
easy to make code twice as fast ala insertion

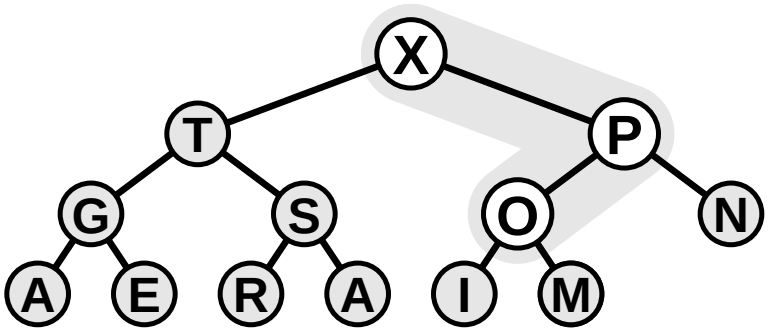
Demotion (sifting down in a heap)

Change key in node at the top of the heap

To restore heap condition:

exchange with larger child if necessary





"Boss's child" model (Royal succession?)

```
fixDown(itemType a[], int k, int N)
   { int j;
      for ( ; k <= N/2; k = j)
         { j = k+k;
            if (j < N && less(a[j], a[j+1])) j++;
            if (!less(a[k], a[j])) break;
            exch(a[k], a[j]);
         }
   }
```

PQ implementation with heaps

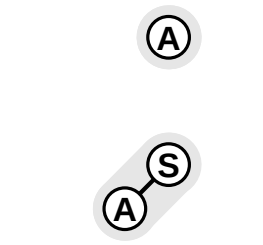
PQinsert: add node at bottom

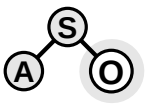
PQdelmax: exch root with node at bottom

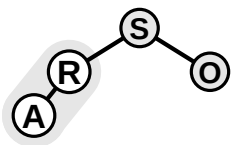
```
static itemType pq[maxPQsize+1];
static int N;

void PQinit() { N = 0; }
int PQempty() { return !N; }
void PQinsert(itemType v)
   { pq[++N] = v; fixUp(pq, N); }
itemType PQdelmax()
   {
      exch(pq[1], pq[N]);
      fixDown(pq, 1, N-1);
      return pq[N--];
   }
```

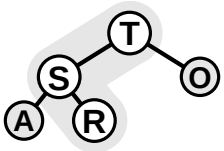
No more than lg N steps (guaranteed)

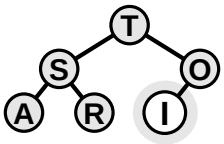
Constructing a heap (top-down)

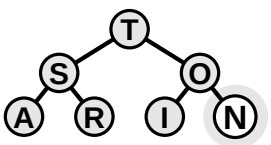


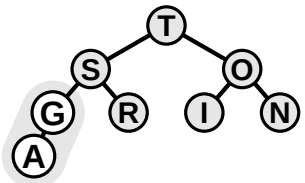


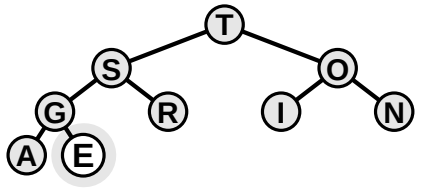


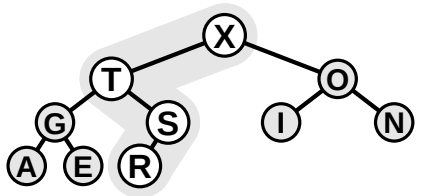


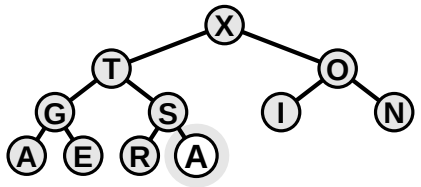


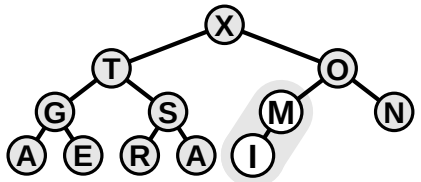


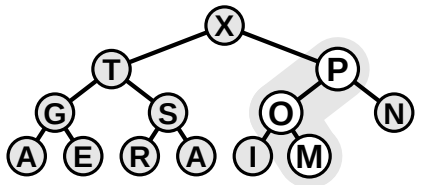


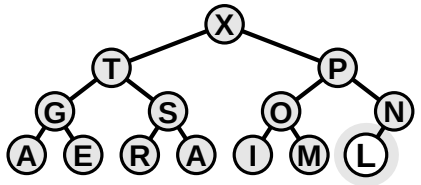


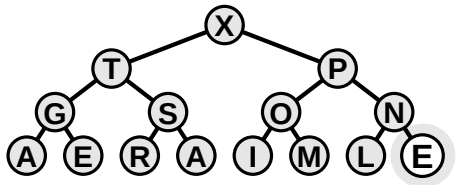




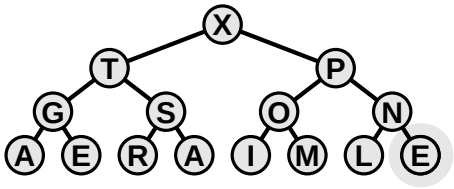


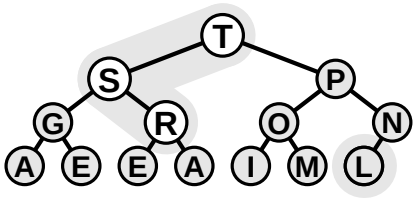


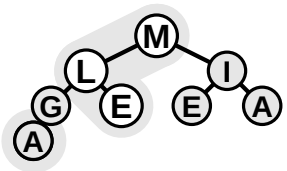


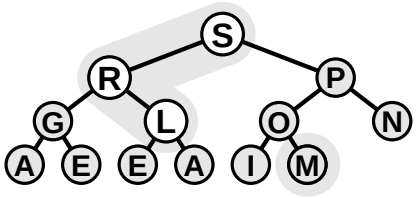


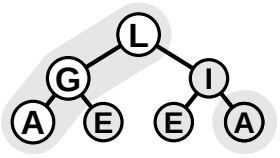
Sorting down a heap

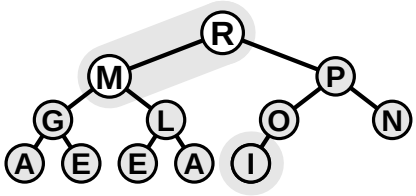


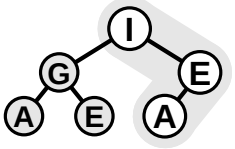


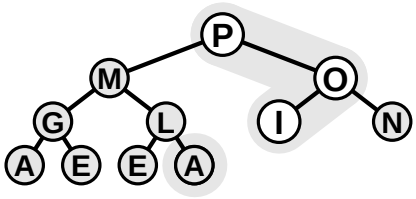


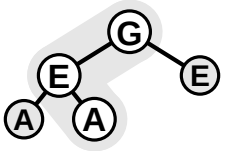


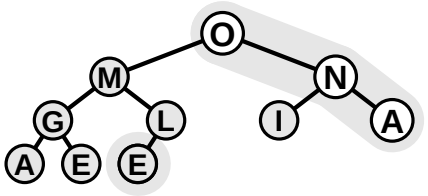


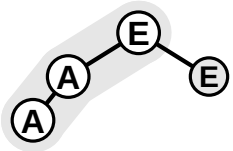


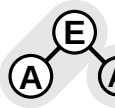


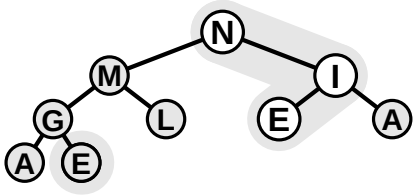


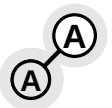










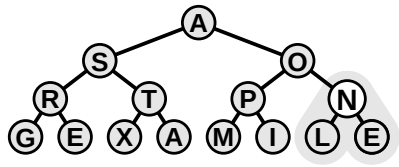


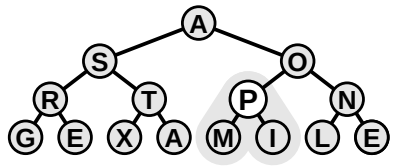
Heapsort

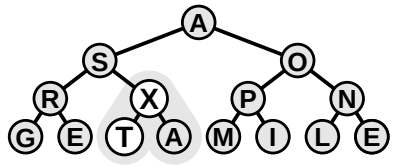
Abandon ADT concept to save space

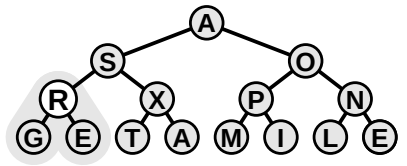
Faster to construct heap from the top down

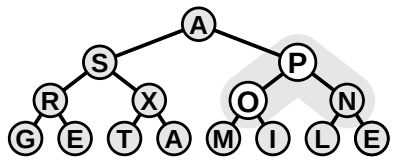
```
#define pq(A) a[l-1+A]
void heapsort(itemType a[], int l, int r)
   { int k, N = r-l+1;
      for (k = N/2; k >= 1; k--)
         fixDown(&pq(0), k, N);
      while (N > 1)
         {
            exch(pq(1), pq(N));
            fixDown(&pq(0), 1, --N);
         }
   }
```

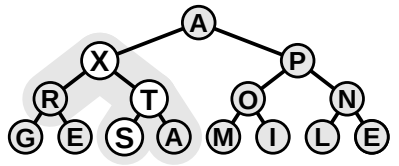


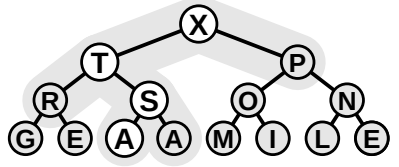












Widely used sorting method

* inplace (no extra space)
* guaranteed NlgN time

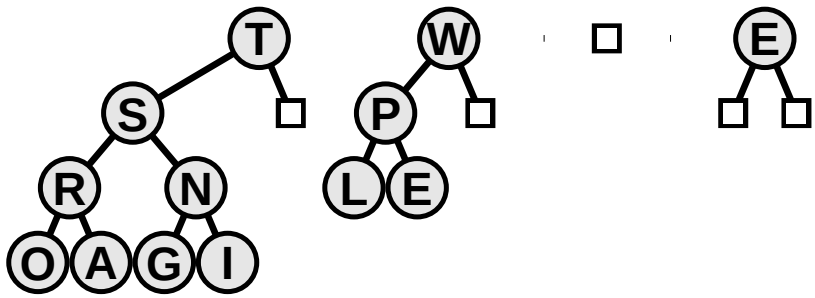
Binomial queues

Support ALL PQ operations in lgN steps
(Heaps have slow merge)

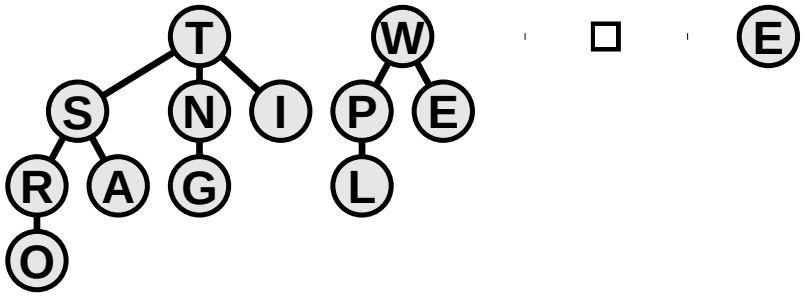
Def: In a LEFT HEAP-ORDERED tree, each node is larger than all nodes in left subtree

Def: A POWER-OF-2 TREE is a binary tree with
* left subtree of root complete
* right subtree empty
(therefore, 2^n nodes)

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



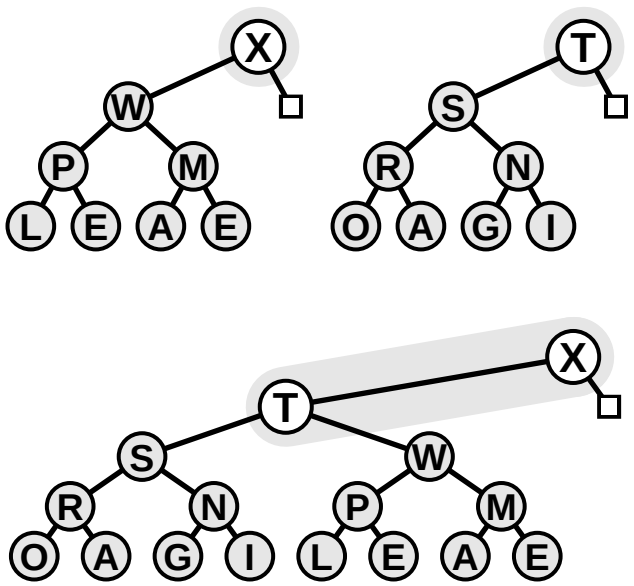
Corresponds to heap-ordered forest:

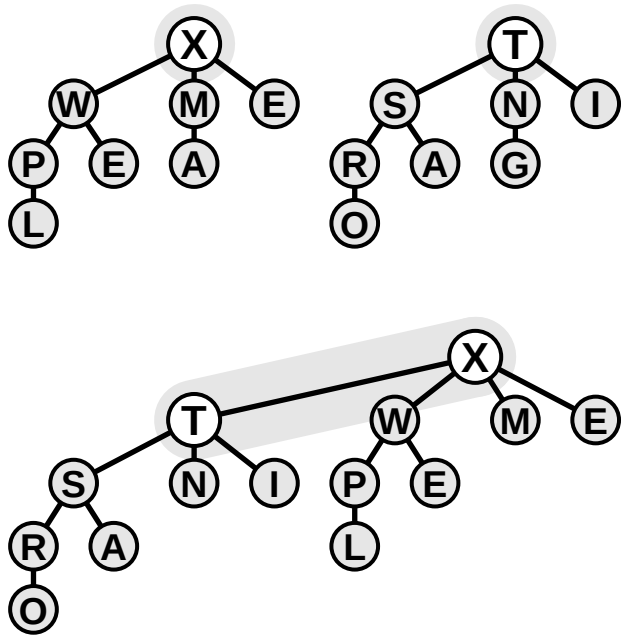


Joining power-of-2 heaps

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 BQnode
  { itemType key; struct BQnode *l, *r; };
typedef struct BQnode* PQhandle;
```

Join operation

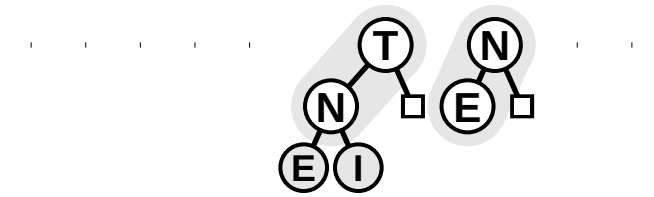
* takes two BQ nodes as input
* returns one BQ nodes as output

```
PQhandle joinPair(PQhandle p, PQhandle q)
  { PQhandle t;
    if (p->key < q->key)
      { t = p; p = q; q = t; }
    q->r = p->l; p->l = q;
    return p;
  }
```

Joining binomial queues

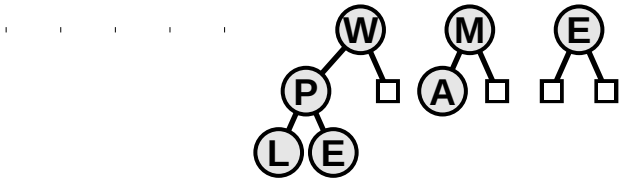
Corresponds to adding binary numbers

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

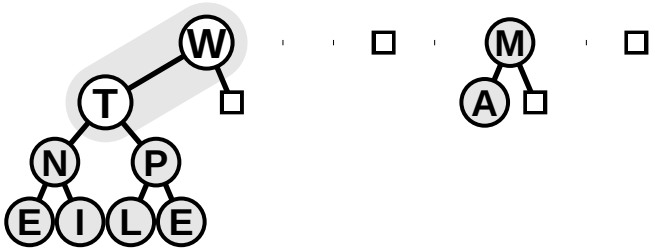


011

111



1010



* lgN time (guaranteed)
* all PQ operations easy to implement with join

Joining binomial queues (code)

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

```
void PQjoin(PQ a, PQ b)  /* a = a+b  */
{ int i; PQhandle c = z;
   for (i = 0; i < maxBQsize; i++)
   if ((b[i] == z) && c == z) ;
   else if ((a[i] != z) && (b[i] != z) && c != z)
      { c = joinPair(c, b[i]); }
   else if ((a[i] == z) && (b[i] == z) && c != z)
      { a[i] = c; c = z; }
   else if ((a[i] == z) && (b[i] != z) && c == z)
      { a[i] = b[i]; }
   else if ((a[i] == z) && (b[i] != z) && c != z)
      { c = joinPair(c, b[i]); a[i] = z; }
   else if ((a[i] != z) && (b[i] == z) && c != z)
      { c = joinPair(c, a[i]); a[i] = z; }
   else if ((a[i] != z) && (b[i] != z) && c == z)
      { c = joinPair(a[i], b[i]); a[i] = z; }
}
```

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 works
for "change priority"

All operations guaranteed lgN

Actually faster, in "amortized" sense

Example: PQinsert N items
* N even, just insert item
* N = ...01, just two steps
* N = ..001, just three steps
...
total cost linear

Basis for modifications getting most operations
down to constant worst-case time
(some of which are practical)

Good candidate for "system" PQ routine