Priority Queues

Priority Queue ADT
Heaps and Heapsort
Binomial Queues

Abstract data types (ADTs)

Separate interface and implementation so as to

- build layers of abstraction
- reuse software

Ex: pushdown stack, FIFO queue

interface: description of data type, basic operations

client: program using operations defined in interface

implementation: actual code implementing operations

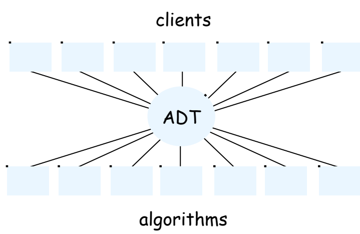
Client can't know details of implementation

- therefore has many implementations to choose from

Implementation can't know details of client needs

- therefore many clients can use the same implementation

ADTs and algorithms



Performance matters!

ADT allows use of better algorithm

(without any change to client)

Idealized scenario

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

Each ADT provides a new level of abstraction

Ex:

client
quicksort
stack
linked list

Total cost depends on

- ADT implementation (algorithm)
- client usage pattern

Might need different implementations for different clients

Basic Priority Queue ADT

Records with keys (priorities)

basic operations

- insert
- remove largest — can substitute smallest for clarity but not both in same client
- create
- test if empty — generic operations common to many ADTs
- destroy
- copy — not needed for one-time use but critical in large systems

Example clients

- simulation
- numerical computation
- data compression
- graph searching — stay tuned

PQ.h

```
void PQinit();
void PQinsert(Item);
Item PQdelmax/min();
 int PQempty();
```

PQ interface in C

PQ example

insert E	E	
insert X	E X	
insert A	E X A	
	E A	remove largest → X
insert M	E A M	
	E A	remove largest → M
insert P	E A P	
insert L	E A P L	
	E A L	remove largest → P
insert E	E A L E	
	E A E	remove largest → L
	A E	remove largest → E
	A	remove largest → E
		remove largest → A

PQ client example

Problem: Find the largest M of a stream of N elements

Example application: Fraud detection (isolate $$ transactions)

Constraint: May not have memory to store N elements

Solution: Use a priority queue

	time	space
elementary PQ	NM	M
heap/BQ	N lgM	M
select	N	N

```
PQinit();
for (k = 0; k < M; k++)
  PQinsert(nextItem());
for (k = M; k < N; k++)
  {
    PQinsert(nextItem());    add next
    t = PQdelmin();          discard smallest
  }
for (k = 0; k < M; k++)
  a[k] = PQdelmin();         M largest
                             left on PQ
```

Ex: top 10,000 in a stream of 1 billion
not possible without good algorithm (also can adapt select)

Unordered-array PQ implementation

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

some other implementations need sentinel

PQ implementations cost summary

Worst-case asymptotic costs for a PQ with N items

	insert	remove max
ordered array	N	1
ordered list	N	1
unordered array	1	N
unordered list	1	N

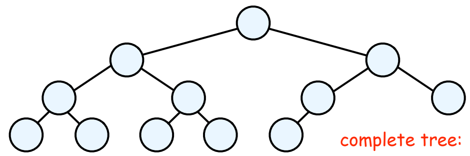
Can we implement both operations efficiently?

Heap

Heap: Array representation of a heap-ordered complete binary tree

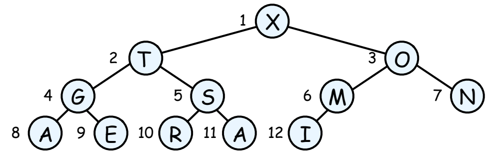
Binary tree

- null or
- node with links to left and right trees



Heap-ordered binary tree

- keys in nodes
- no smaller than children's keys



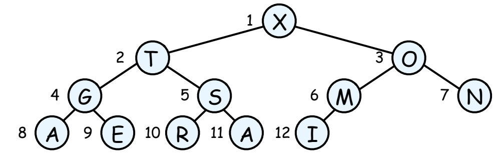
Array representation

- take nodes in level order
- no explicit links

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

Heap properties

Largest key is at root



Can use array indices to move through tree

- parent of node at k is at k/2
- children of node at k are at 2k and 2k+1

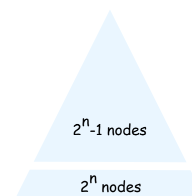
1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

Length of path in N-node heap is at most ~lg N

n levels when $2^n \le N < 2^{n+1}$

$n \le \lg N < n+1$

~lg N levels



Promotion (bubbling up) in a heap

Suppose that a node at the bottom is larger than its parent

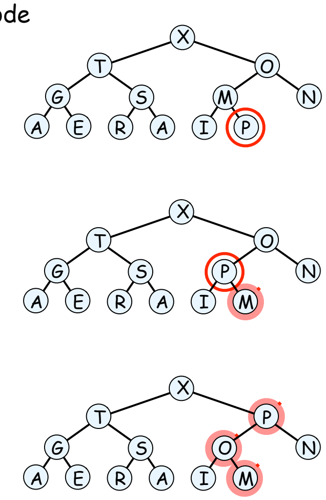
Invariant: Heap condition violated only at that node

To eliminate the violation

- exchange with parent
- maintains invariant (why?)
- moves up the tree
- continue until node not larger than parent

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

parent of node at k is at k/2



Peter principle:
node rises to level of incompetence

1	2	3	4	5	6	7	8	9	10	11	12	13
X	T	O	G	S	M	N	A	E	R	A	I	P
X	T	P	G	S	O	N	A	E	R	A	I	M

Demotion (sifting down) in a heap

Suppose that a node at the top is smaller than a child

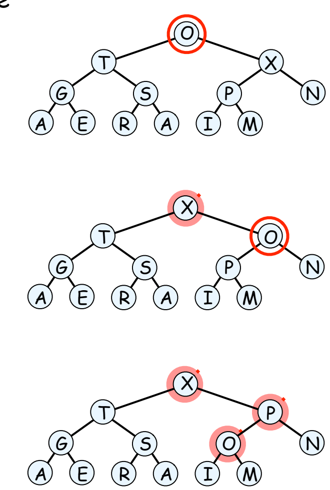
Invariant: Heap condition violated only at that node

To eliminate the violation

- exchange with larger child
- maintains invariant (why?)
- moves down the tree
- continue until node not smaller than children

```
sink(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;
      }
  }
```

children of node at k are at 2k and 2k+1



Power struggle: better subordinate promoted

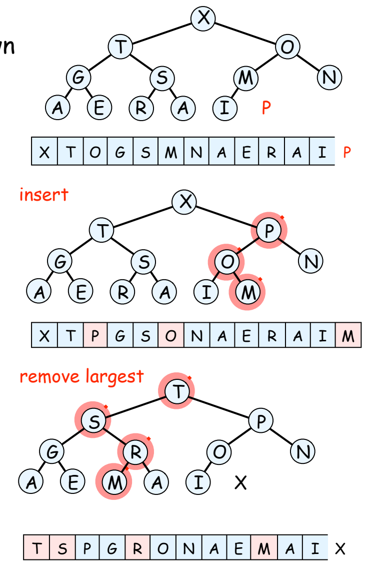
1	2	3	4	5	6	7	8	9	10	11	12	13
O	T	X	G	S	P	N	A	E	R	A	I	M
X	T	P	G	S	O	N	A	E	R	A	I	M

Heap-based PQ implementation

insert
  add node at end, then promote
remove largest
  exchange root with node at end, then sift down

```
static Item *pq;
static int N;
void PQinit(int maxN);
int PQempty();
PQinsert(Item v)
  { pq[N++] = v; swim(pq, N); }
Item PQdelmax()
  {
    exch(pq[1], pq[N]);
    sink(pq, 1, N-1);
    return pq[N--];
  }
```

same as elementary array-based



PQ implementations cost summary

Worst-case asymptotic costs for a PQ with N items

	insert	remove max
ordered array	N	1
ordered list	N	1
unordered array	1	N
unordered list	1	N
heap	lg N	lg N

Digression: Heapsort

First pass: build heap
  add item to heap at each iteration, then sift up
  (or can use faster bottom-up method; see book)
Second pass: sort
  remove maximum at each iteration
  exchange root with node at end, then sift down

```
#define pq(A) a[L-1+A]
void heapsort(Item a[], int L, int R)
  { int k, N = r-l+1;
    for (k = 2; k <= N; k++)
      swim(&pq(0), k);
    while (N > 1)
      { exch(pq(1), pq(N));
        sink(&pq(0), 1, --N);
      }
  }
```

build heap

remove maximum; sift down

in the heap

not in the heap

E	X	A	M	P	L	E
E	X	A	M	P	L	E
X	E	A	M	P	L	E
X	E	A	M	P	L	E
X	M	A	E	P	L	E
X	P	A	E	M	L	E
X	P	L	E	M	A	E
X	P	L	E	M	A	E
P	M	L	E	E	A	X
M	E	L	A	E	P	X
L	E	E	A	M	P	X
E	A	E	L	M	P	X
E	A	E	L	M	P	X
A	E	E	L	M	P	X
A	E	E	L	M	P	X

Significance of Heapsort

Q: Is there a sort that uses
- O(N log N) running time in the worst case and
- no extra memory ?

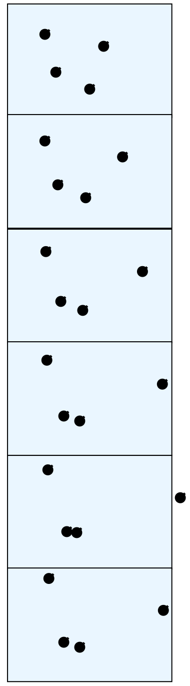
A: Yes. Heapsort.

Not mergesort?
- O(N) extra space
- (challenge for the bored: design an inplace merge)

Not quicksort?
- quadratic in worst case (but probabilistic guarantee is as good)
- O(log N) extra space (not an issue in practice)

Heapsort is OPTIMAL for both time and space, BUT
- inner loop longer than quicksort's
- makes poor use of cache memory

Event-based simulation



Challenge: Animate N moving particles

- each has given velocity vector
- bounce off edges, one another on collision

Example applications: molecular dynamics, traffic, ...

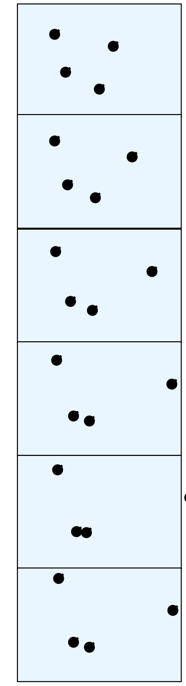
Naive approach: t times per second

- update particle positions
- check for collisions, update velocities
- redraw all particles

Problems:

- N^2t collision checks per second
- may miss collisions

PQ for event-based simulation



Approach: Use PQ of events with time as key

- put collision event on PQ for each particle (calculate time of next collision as priority)
- put redraw events on PQ (t per second)

Main loop: Remove next event from PQ

- redraw: update positions and redraw
- collision: update velocity of affected particle(s) and put new collision events on PQ

More PQ operations needed:

- may need to remove items from PQ
- may want to join PQs for different sets of events (Ex: join locals to national for air traffic control)

More sophisticated PQ interface needed

Extending the Priority-Queue ADT

Records with keys (priorities)

Full set of operations

- create
- test if empty
- destroy
- copy

generic operations for first-class ADTs

- insert
- remove largest

operations that characterize PQs

- remove
- find largest
- change key
- join

other operations that many clients need

New operations complicate the interface

- need to refer to items in PQ for remove, change key
- need to refer to PQs for destroy, copy, and join
- while still maintaining separation between client and implementation

Object-oriented programming (OOP)

Extended Priority-Queue ADT

Records with keys (priorities)

Full set of operations

- create
- test if empty
- destroy
- copy

generic operations for first-class ADTs

- insert
- remove largest

operations that characterize PQs

- remove
- find largest
- change key
- join

other operations that many clients need

pointers to structures to be specified in implementation (Read Sections 4.8 and 4.9)

PQfull.h

```
typedef struct pq* PQ;
typedef struct PQnode* PQlink;
     PQ PQinit();
    int PQempty(PQ);
 PQlink PQinsert(Item,PQ);
   Item PQdelmax(PQ);
   void PQchange(PQ,PQlink,Item);
   void PQdelete(PQ,PQlink);
     PQ PQjoin(PQ,PQ);
```

First-class PQ interface in C

Handle implementation in C: use pointers to unspecified structures

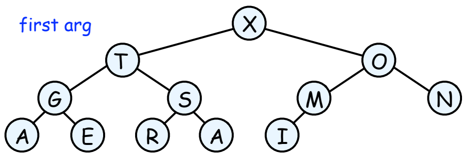
- a PQ is a pointer to a pq struct
- a PQlink is a pointer to a PQnode struct
- no way for client to know pq and PQnode implementations

Note: solution easier in OOP languages like Java and C++ because primitives are built in

PQ challenge: join two heaps

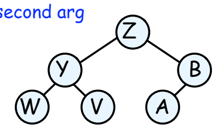
```
PQ PQjoin(PQ a, PQ b)
```

first arg



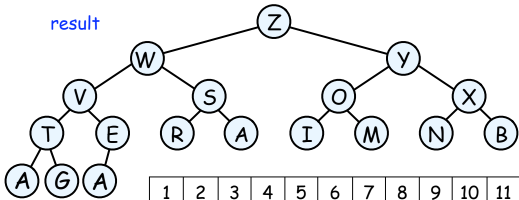
1	2	3	4	5	6	7	8	9	10	11	12
X	T	O	G	S	M	N	A	E	R	A	I

second arg



1	2	3	4	5	6
Z	Y	B	W	V	A

result



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Z	W	Y	V	S	O	X	T	E	R	A	I	M	N	B	A	G	A

Would it help to use linked structures?
Hard to beat trivial algorithm (rebuild the whole heap)

First-class PQ implementations cost summary

New operations introduce new algorithmic challenges

	insert	remove max	remove	find max	change key	join
ordered array	N	1	N	1	N	N
ordered list	N	1	1	1	N	N
unordered array	1	N	1	N	1	N
unordered list	1	N	1	N	1	1
heap	lg N	lg N	lg N	1	lg N	N

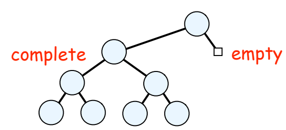
Can we implement all the operations efficiently?

Binomial Queue

Binomial queue with N nodes: forest of left-heap-ordered power-of-2 trees, one for each term in the binary decomposition of N

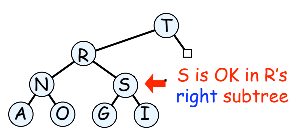
power-of-two tree (pott): binary tree with

- empty right subtree
- complete left subtree



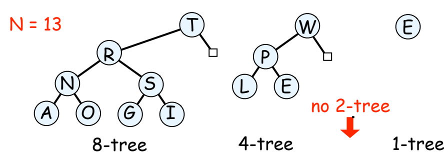
left-heap-ordered pott (lhopott)

- key in each node
- no smaller than all keys in left subtree

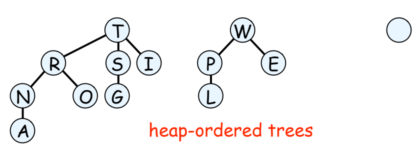


binary decomposition:

- sum of distinct powers of 2
- direct from binary representation Ex: $13 = 1101_2 = 8 + 4 + 1$

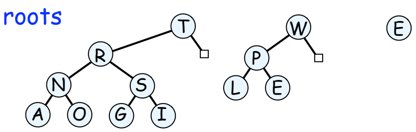


lhopott is binary-tree representation of heap-ordered general tree



Binomial queue properties

Largest key is at one of the roots



Can use links to move down tree

- two links per node

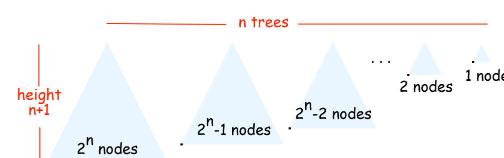
```
struct PQnode
    { Item key; PQlink l, r; };
struct pq { PQlink *bq; };
```

~lg N trees in N-node BQ

- ~lg N links to represent BQ

Length of path in N-node BQ is at most ~lg N

path length in 2^n-tree is (n+1)

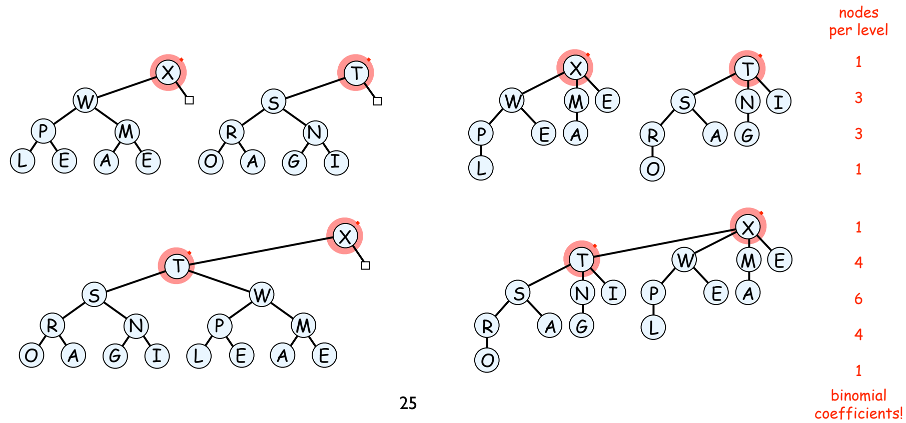


Joining two equal-sized lhopotts

A constant-time operation

- take larger of two roots as root
- combine other root, two subtrees to make complete lho left subtree
- result is lho if arguments are lho

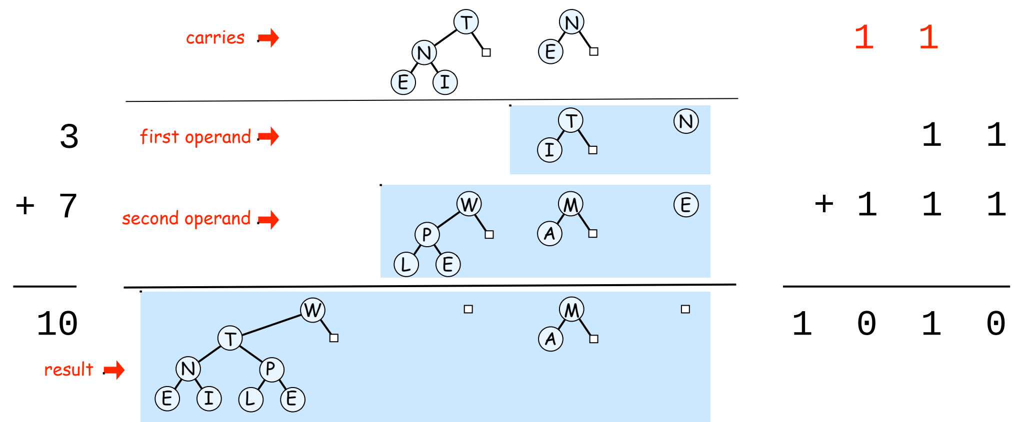
```
PQlink pair(PQlink p, PQlink q)
  { PQlink t;
    if (less(p->key, q->key))
      { p->r = q->l; q->l = p; return q; }
    else
      { q->r = p->l; p->l = q; return p; }
  }
```



Joining two binomial queues

Mimic addition of corresponding binary numbers

- adding 1 bits corresponds to joining equal-sized lhopotts
- 1+1 = 10 or 1+1 + 11 corresponds to carry
- result is a BQ whose size is sum of operand sizes



Joining two binomial queues (code)

Not much more difficult than binary addition!

```
#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;
        }
  }
```

carry

case	c	b	a		a	c
0	0	0	0		a	0
1	0	0	1		a	0
2	0	1	0		b	0
3	0	1	1		0	a+b
4	1	0	0		c	0
5	1	0	1		0	a+c
6	1	1	0		0	b+c
7	1	1	1		a	b+c

result

BQ-based PQ implementation

Join provides basis for all the implementations

insert:

- join singleton BQ

remove maximum:

- scan roots to find max, remove its tree
- join children of max with rest of BQ

change priority:

- demote, promote as with heaps

remove:

- replace removed node with max in its tree
- join children of max with rest of BQ

PQ implementations cost summary

Worst-case asymptotic costs for a PQ with N items

	insert	remove max	remove	find max	change key	join
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

Priority Queues: Summary

Algorithm-design success story

PQ ADT

- identifies a useful computational abstraction

Heap

- provides efficient implementations of basic operations

Binomial queue

- provides efficient implementations of all operations

Ingenenious fundamental data structures

Surprising fact: there is still room for improvement!

PQ implementations cost summary

Worst-case asymptotic costs for a PQ with N items

	insert	remove max	remove	find max	change key	join
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

Algorithms have been invented that meet these bounds,
BUT it is difficult to beat BQs in practice