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

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

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
• create
• test if empty
• destroy
• copy

Example clients
• simulation
• numerical computation
• data compression
• graph searching

void PQinit();
void PQinsert(Item);
Item PQdelmax/min();
int PQempty();
PQ interface in C
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

<table>
<thead>
<tr>
<th>PQ client example</th>
<th>PQ example</th>
</tr>
</thead>
</table>
| Problem: Find the largest $M$ of a stream of $N$ elements | \begin{align*}
\text{insert } E \rightarrow & \quad E \\
\text{insert } X \rightarrow & \quad E \quad X \\
\text{insert } A \rightarrow & \quad E \quad A \\
\text{insert } M \rightarrow & \quad E \quad A \quad M \\
\text{insert } P \rightarrow & \quad E \quad A \quad P \\
\text{insert } L \rightarrow & \quad E \quad A \quad P \quad L \\
\text{insert } E \rightarrow & \quad E \quad A \quad L \quad E \\
\end{align*} |

| \text{remove largest } \rightarrow \quad X | \text{remove largest } \rightarrow \quad M |
| \text{remove largest } \rightarrow \quad P | \text{remove largest } \rightarrow \quad L |
| \text{remove largest } \rightarrow \quad E | \text{remove largest } \rightarrow \quad A |

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

PQ implementations cost summary

<table>
<thead>
<tr>
<th>PQ implementations cost summary</th>
<th>PQ example</th>
</tr>
</thead>
</table>
| Worst-case asymptotic costs for a PQ with $N$ items | \begin{align*}
\text{insert} & \quad \text{time} & \text{space} \\
\text{ ordered array} & \quad N & 1 \\
\text{ ordered list} & \quad N & 1 \\
\text{ unordered array} & \quad 1 & N \\
\text{ unordered list} & \quad 1 & N \\
\end{align*} |

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

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

Length of path in N-node heap is at most \( \sim \log N \)

n levels when \( 2^n \leq N < 2^{n+1} \)

\( n \leq \log N < n+1 \)

\( \sim \log N \) 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

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

Peter principle:
node rises to level of incompetence

Power struggle: better subordinate promoted
Heap-based PQ implementation

**insert**
- add node at end, then promote

**remove largest**
- exchange root with node at end, then sift down

```c
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--];
    }
```

---

**PQ implementations cost summary**

Worst-case asymptotic costs for a PQ with N items

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>remove max</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>unordered list</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>heap</td>
<td>(\lg N)</td>
<td>(\lg N)</td>
</tr>
</tbody>
</table>

---

**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

---

**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

```c
#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);
        }
    }
```
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:
- \( N^2 t \) collision checks per second
- may miss collisions

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
- insert
- remove largest
- remove
- find largest
- change key
- join

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
- insert
- remove largest
- remove
- find largest
- change key
- join

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)

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 PQJoin(PQ a, PQ b)

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 = 1101₂ = 8 + 4 + 1

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

First-class PQ implementations cost summary

New operations introduce new algorithmic challenges

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>remove max</th>
<th>remove min</th>
<th>find max</th>
<th>change key</th>
<th>join</th>
</tr>
</thead>
<tbody>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>N</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered list</td>
<td>N</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>1</td>
<td>N</td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td>unordered list</td>
<td>1</td>
<td>N</td>
<td>1</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>heap</td>
<td>(\lg N)</td>
<td>(\lg N)</td>
<td>(\lg N)</td>
<td>1</td>
<td>(\lg N)</td>
<td>N</td>
</tr>
</tbody>
</table>

Can we implement all the operations efficiently?

Can use links to move down tree
  • two links per node
  • \(\sim \lg N\) trees in N-node BQ
  • \(\sim \lg N\) links to represent BQ

Length of path in N-node BQ is at most \(\sim \lg N\)

Path length in \(2^n\)-tree is \((n+1)\)
Joining two equal-sized lhoptts

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

Joining two binomial queues

Mimic addition of corresponding binary numbers
- adding 1 bits corresponds to joining equal-sized lhoptts
- $1+1 = 10$ 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!

```c
#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 0: break;
            case 1: c = 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;
        }
}
```

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Insert</th>
<th>Remove</th>
<th>Find Max</th>
<th>Change Key</th>
<th>Join</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Heap</strong></td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$1$</td>
<td>$\log N$</td>
</tr>
<tr>
<td><strong>Binomial Queue</strong></td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$\log N$</td>
</tr>
<tr>
<td><strong>Best in Theory</strong></td>
<td>$1$</td>
<td>$\log N$</td>
<td>$\log N$</td>
<td>$1$</td>
<td>$1$</td>
</tr>
</tbody>
</table>

---

### 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

**Ingenious fundamental data structures**

Surprising fact: there is still room for improvement!

---

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