Temporal Data Structures

Data structures which make time-travel

Lijie Chen

Tsinghua Institute for Interdisciplinary Information Sciences

What is temporal data strucutures

Data structures which make time-travel

- persistent data structures
- retroactive data structures

Pointer Machine

An abstract computational machine model.
In this model we think of data structures as collections of nodes of a bounded size with entries for data. Each piece of data in the node can be either actual data, or a pointer to a node.
Just like struct in C++, record in Pascal, we can store actual data or pointer in a node.

How to work in pointer machine

Unlike RAM model, you can't access variable directly, but only though series of pointer.
A virtual root has pointer to some useful thing.
Access in pointer Machine:
We have a accessed set node set S, each time we can pick $u \in S$ and access v if u has a pointer of v, and add v into set S.
Modification in pointer Machine:
During access, we can modify some attribute of node in accessed set S.

How to work in pointer machine

So the primitive operation is:

1. `x = new Node()`
2. `x = y.field`
3. `x.field = y`
4. `x = y + z`, etc (i.e. data operations)
5. `destroy(x)` (if no other pointers to `x`)

Why pointer machine?

Everything is based on pointer, persistent is very easy.
We can stimulate other model(like RAM) with some overhead.

Segment tree in Pointer Machine

```
struct SegmentTree {
    SegmentTree*pl, *pr;
    int l, r, opt;
};
```

Pointer Machine

Many data structures can be described in point machine.
But variable size data structures like array can't. But we can using a binary tree to implement array, with a overhead of $O(\log n)$.
Later we will talk about fully persistent array in $O(\log \log n)$.
Which is also the lower bound.
The everything we will talk about below will using pointer machine paradigm by default.

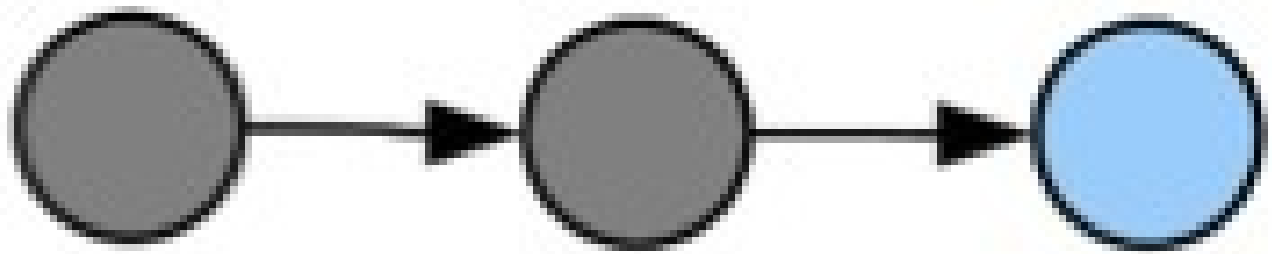
Persistent data structures

- Partial Persistence
- Full Persistence
- Confluent Persistence
- Functional Persistence

Specific persistent data structures like segment trees, binary search trees are well studied in OI.
The general method to make every data structures persistent.
If the data structure has bounded in-degree which is $O(1)$, then it can be made partial persistent in worst case $O(1)$. And can be made fully persistent in amortized time $O(1)$.
To better illustrate the category of persistence, we can use version graph.

Partial Persistence

What is partial persistence?
Modification only occur now, query about the past.

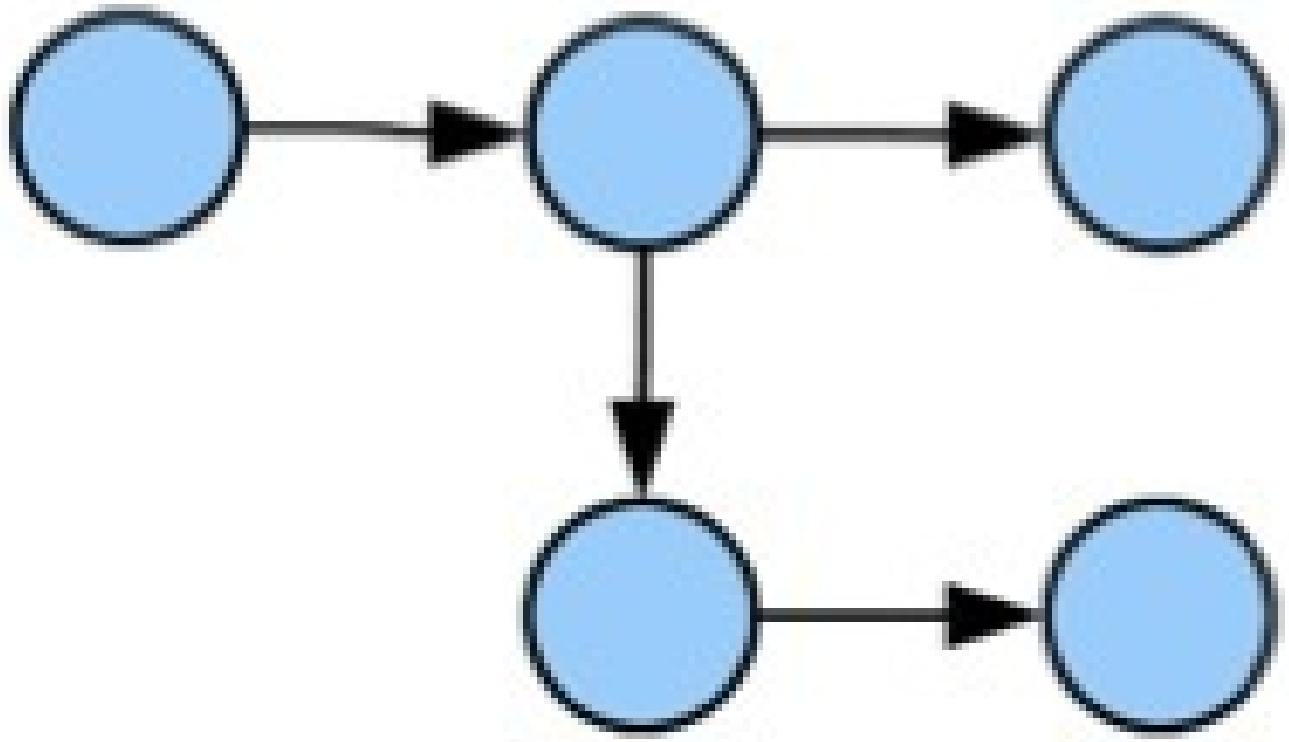


(a) Partial.

Full Persistence

What is Full persistence?

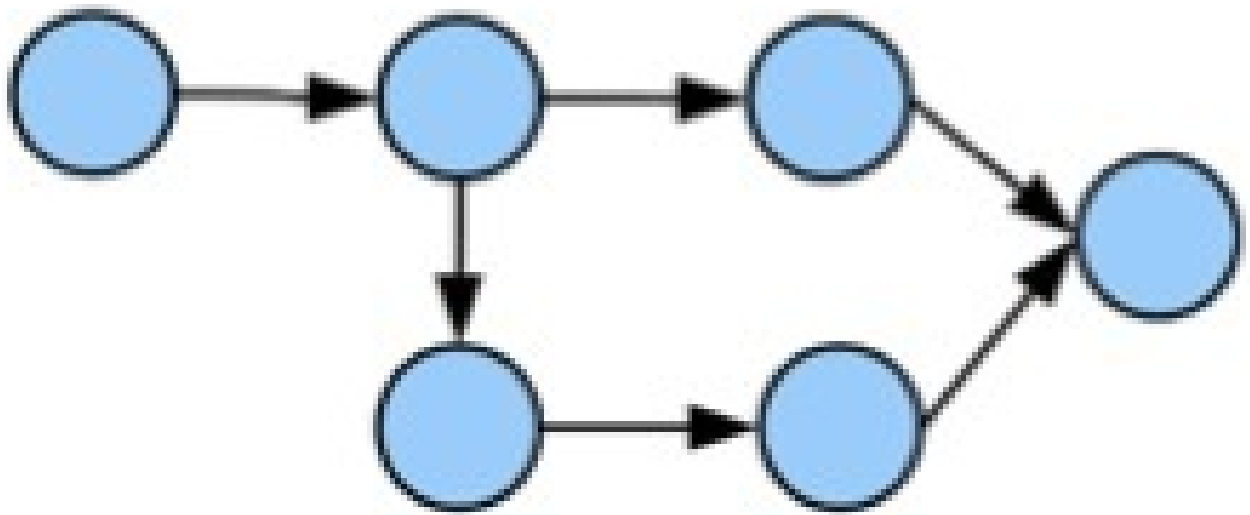
Modification about the past, query about the past.



(b) Full

Confluent Persistence

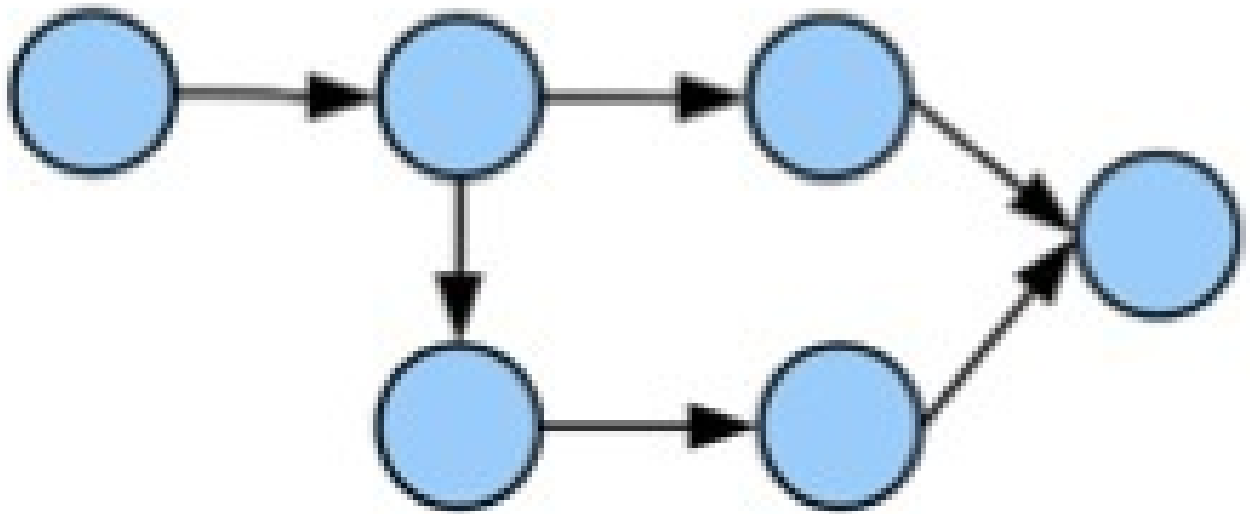
What is Confluent Persistence?
In addition to full persistence, we can even combine two previous version into one.



(c) Confluent/ Functional

Functional Persistence

What is Functional Persistence?
We restrict the way of implementing persistent data structure.
We can not modify, we can only create new node if we need.
Functional way.



(c) Confluent/ Functional

The general method to make data structures partial persistent in a pointer machine

Every data structures in pointer machine, its function can be recognized as a sequence of write and read.
So In partial persistent data structures, we have:

- read(node u,version v,variable i): read the variable i in the version v of node u.
- write(node u,variable i,value var): write value var to the variable i of node u.

When the in-degree is bounded by $O(1)$, it can be done in worst case $O(1)$.

The general method to make data strucutres fully persistent in a pointer machine

Every data structures in pointer machine, its function can be recognized as a sequence of write and read.
So In fully persistent data structures, we have:

- read(node u,version v,variable i): read the variable i in the version v of node u.
- write(node u,version v,variable i,value var): write value var to the variable i in the version v of node u. which return a new version.

When the in-degree is bounded by $O(1)$, it can be done in amortized case $O(1)$.

Stimulate the RAM model

As RAM can be seen as a big array. If we can stimulate the fully persistent array in $O(g(n))$ time, then every data structure take $O(f(n))$ time can be made full persistent in $O(g(n)f(n))$ time. Actually fully persistent array can be solved int $O(\log \log n)$ time. which is also a lower bound.

van Emde Boas Tree

van Emde Boas Tree is a data structure which support fast predecessor query of integer.
Suppose we deal with integer in $[0, u)$.

- * insert: $\log \log n$
- * predecessor(x), successor(x): $\log \log n$

van Emde Boas Tree

Denote $V(u)$ to be the data structure which integer in $[0, u)$, And $T(u)$ to be the time complexity fro $V(u)$.
We can show $T(u) \leq T(\sqrt{u}) + O(1)$
First we divide $[0, u)$ into $\sqrt{u}$ block, let $V(u).block[i]$ denote the $i - th$ block.
And we keep the minimum value as $V(u).min$, and add others value into the blocks and summary(not the minimum). Also we record $V(u).max$.
Also let $V(u).summary$ to store the index of every non-empty block.
Then we can see that block and summary can be seen as instance of $V(\sqrt{u})$.
We can use hash to get the block with index.

van Emde Boas Tree:Insertion

Let insertion time to be $T(u)$.
Suppose we insert x, let $x = a\sqrt{u} + b$.
If $V(u)$ is empty, then we let $V(u).min = x$, and we are done.
Otherwise we check x with $V(u).min$ to maintain the minimum value, and insert b(or the old minimum's b) to $V(u).block[a]$ and a to $V(u).summary$.
We can see $T(u) = T(\sqrt{u}) + O(1)$
So $T(u) = O(\log \log n)$

van Emde Boas Tree:Query

Suppose we Query x's predecessor at $V(u)$, let $x = a\sqrt{u} + b$.
If $V(u).block[a]$ exist, we check whether x's predecessor is in it using its min and max. If so, we query in V(u).block[a].
Otherwise, we query in $V(u).summary$ to get the predecessor block of a, and return its max.
We can see that the time complexity is $O(\log \log n)$ again.

Fully persistent array

We maintain the version tree.
Maintain the ET of it.
For each array index i, we denote L_i as the nodes in version tree where $A[i]$ has been changed.
We need to find the lowest ancestor of u which in L_i.
If we order L_i by their position in ET, it is exactly the predecessor problem.

Labels

But we need integer label in VEB tree solution.
And as we can insert a label in ET sequence, things get complicated.
Now we need a data structure, which support:

- * insert(x,t) insert a element t right after element x.
- * each element has a integer label which preserve their order.

weight BST

We know that weight BST can give each element a label of $O(\log n)$ bits which preserve their order and support insert in $O(\log n)$ time.

Labels

For better performance, we partition the sequence in many blocks, each blocks have size $O(\log n)$.
For each block, we use weight BST to give the whole block a label of $O(\log n)$ bits using this block's minimum element.
Inside a block, we use brute force to give each element in a block a label of $O(\log n)$ bits.
When a block have size $> 2 \log n$, we split it in half.

Labels

Seems good, but if our operation change a block's minimum element. we need to relabel $O(\log n)$ elements. which need $O(\log n \log \log n)$ time, which is even worse.

Bucketing

When finding predecessor, we partition the sequence in blocks of size $O(\log^2)$.
Inside a block, we use a BST to find the predecessor, which need $O(\log \log n)$ time.
And we use a VEB tree to maintain each block's order using each block's minimum element's label.
Then we only need the label of each block's header.
There is $O(n/\log^2 n)$ such headers, as one insert might cause $O(\log n)$ element's label to be changed, we might expect $O(\log^{-1} n)$ header's label change on average.

Weighted Label Maintainance

But as the header may be nonuniform.
We can changed the previous labeling method to be based on header weight(A header has weight 1, other has weight 0).
Then we can have a $O(\log \log n)$ solution for fully persistent array.
This means, we can make EVERY normal data structure with worst case complexity $O(f(n))$ fully persistent in $O(f(n) \log \log n)$ time.

Open Questions

De-amortization of full persistence

Is there a matching lower bound for both full and partial persistence?

Functional Tries

Why Functional Tries is important?
We can see file system as a trie, which leaf node represent file, internal node represent folder, edge name represent file name or folder name.
Then Functional Tries can be used to represent version of the file system.

Functional Tries:Naive approach

As it is a tree, we can simply use the path-copying method.
For a tree which has tree depth d, the running time is $O(d)$.
What if the tree is very deep?

Functional Tries:Motivation

In real work, we often do some work in one folder, then switch to its parent folder, and do some other work, and switch back, and do some other work.
If we employ the path-copying and we the tree has huge deep, it could be very slow.

Functional Tries:The Concept of Finger

To abstract, we can see in previous example, where we do the job don't vary much.
We can say that we have some "finger", each finger point to some node in the Trie. And we can only do modification at where the finger stand.
At the previous example, we just maintain one finger, and each time move this finger to neighbouring node.

Functional Tries:The Concept of Finger

Using this concept, we can do much faster.
Let the maximum degree of the node to be Δ. And there is n nodes. And suppose we only have $O(1)$ amount of finger.
Then we support functional trie which finger move take $O(\log \Delta)$ time, and modification take $O(\log \Delta)$ time.
If we just want fully persistent, then we can even have both $O(\log \log \Delta)$ time.

Functional Tries

Suppose there is k fingers, denote them by F.
Get the Steiner tree of those fingers, which will have at most $2k$ vertices after compressing every 2-degree node. denote those node by PF.
We maintain the compressed tree represented by PF. For each edge on it, it is actually a path in the original trie. We denoted such a path as a tendon. For each tendon, we represent it by a functional dequeue, the elements in this dequeue is the vertices on that path.
For each vertex, we maintain a functional BST of depth $O(\log \Delta)$, which represent the other adjacent vertices of it.
We use a functional BST to store every finger by their node-id. For each finger, we use a functional BST of depth $O(1)$ to store the adjacent tendon, and a functional BST of depth $O(\log \Delta)$ to store other neighbor.

Functional Tries: Movement

How to perform operation on this?
Discussion.

Functional Dequeue

How to make functional dequeue?
Discussion.

Functional Dequeue

Let dequeue $D = P + M + S$. P, S are two buffer which have length at most 3.
M is again a dequeue itself.
We may change D's representation tripe(for example take the last element o P and put it in front of M). But won't change its content.
PushLeft,popLeft works trivial if P is not full and empty.
Otherwise recursive do it in M.
Potential analysis show it is amortized $O(1)$.

Functional Link-Cut Tree

The worst case $O(\log n)$ algorithm described in the original paper is good for functional.
Why the amortized one doesn't work?

Refresh of that method

Decompose the trie into a set of heavy paths, and represent each heavy path by a globally biased binary tree.
Tied together into one big tree which we call the representation tree.
An edge is heavy if more than half of the descendants of the parent are also descendants of the child. A heavy path is a contiguous sequence of heavy edges.
Because any node has at most one heavy child, we can decompose the trie into a set of heavy paths, connected by light (non-heavy) edges.

Refresh of that method

The weight w_v of a node v is 1 plus the number of descendants of v through a light child of v.
Global biased binary tree will make the total w_v of both children balanced. So the representation tree is of $O(\log n)$ depth.

Why path-coping does not work

For the expose operation to work, for each heavy-path's head, we should store its parent.
But it is bad for path copying.
Why?

Finger

Again, we introduce finger here too.
For each relevant vertex, we store a finger, which store all its ancestor in the represent tree.
For each vertex, we make an auxiliary globally biased tree of its light children.
We can use catenable functional dequeue of dequeue to store each finger.
It can support the parent operation.

Retroactive Data Structures

What is retroactive data structures?
We can query about the current version of the data structures, while we can insert or delete the modifications in the past!

- Partial retroactivity
- Full retroactivity
- Nonoblivious retroactivity

Partial retroactivity

Query about the current version of the data structures, while we can insert or delete the modifications in the past.

- Insert(t,modification): insert a modification(which is an original modification) at time t.
- Delete(t,modification): Delete the modification at time t.
- query: query the current version.

Fully retroactivity

Query about the every version(current or the past) of the data structures, while we can insert or delete the modifications in the past.

- Insert(t,modification): insert a modification(which is an original modification) at time t.
- Delete(t,modification): Delete the modification at time t.
- Query(t,query) : query the version at time t.

Nonoblivious retroactivity

Now we even put the query into the sequences.
For each change we made, we want to know the first affected query.

A example of priority queue

Initially it is empty.
insert 1 insert 2 delete-min insert 3 : current one is [2,3]
if we change the insert 2 into insert 0
insert 1 insert 0 delete-min insert 3 : current one is [1,3]

How to achive this?

The bad news is that it is indeed impossible to change every data structures into their retroactive version.
Suppose we have a data structures, which hold two variables X, Y which are initial 0.
And it support the following operation.

- $X = a$
- $Y = Y + a$
- $Y = X \cdot Y$
- query the value of Y

We make the following modification sequence:
$X = x\ Y = Y + a_n\ Y = X \cdot Y\ Y = Y + a_{n-1}\ \cdots\ Y = Y + a_0$.
Then we know the answer is $\sum_{i=0}^{n} a_i x^i$.
If we change the first operation. Then we need to evaluate this polynomial for another x.

How to achive this?

In history-independent algebraic decision tree, for any field, independent of pre-processing of the coefficients, need $\Omega(n)$ field operations (result from 2001), where n is the degree of the polynomial.
So it takes at least $\Omega(n)$ time. We can't do better than brute force.

How to achive this?

But good news is that we can make many data structures retroactive.

- queue: $O(1)$ partial $O(\log n)$ full
- deque: $O(\log n)$ full
- decomposable search problem:
- priority queue: $O(\log n)$ partial

Commutative and Inversion operation

- commutative: $a, b \Leftrightarrow b, a$
- inversion: $\forall a, \exists\ a^{-1}\ s.t.\ a, a^{-1} \Leftrightarrow do\ nothing$

Then $Insert(t, modification) \Leftrightarrow Insert(now, modification)$
Then $Delete(t, modification) \Leftrightarrow Modify(now, modification^{-1})$

Queue

Operation:

- enqueue(x): put x at the end of the queue
- dequeue(): remove the first element

Query:

- front(): return the first element of the queue
- back(): return the last element of the queue

Time complexity:

- partial: $O(1)$
- full: $O(\log n)$

Queue:Partial

How to make queue partial retroactive?
Discussion.

Queue:Partial

Quite trivial, we can use a double linked list to maintain every enqueue operation.
Then we know that the current version is a continuous sub-sequence of it.
Maintain two pointer: F, B point to the front and the back of the queue.
Update is trivial.

Queue:Full

How to make queue fully retroactive?
Discussion.

Queue:Full

Due to the property of queue. It is still not hard.
We maintain two balanced binary tree T_e, T_d, store the enqueue and dequeue operation ordered by time respectively.

- Query(t,front): first count the number of dequeue operation at time $\leq t$. Denote it by d. Then return the $d+1$-th elements in T_e.
- Query(t,back): return the last element in T_e which inserted time $\leq t$.

$O(\log n)$

Deque:Fully

How to make queue fully retroactive?
Discussion.

Deque

Operation:

- pushL(x),pushR(x): put x at the left(right) of the queue
- popL(),popR(): remove the first(last) element

Query:

- front(): return the first element of the queue
- back(): return the last element of the queue

Time complexity:

- full: $O(\log n)$

Deque:Full

In a array-based deque implementation, we use two index L, R, and $A[L...R]$ of the array is our deque.

- initially: L=1,R=0
- pushL(x): L=L-1,A[L]=x
- popL: L=L+1
- pushR(x): R=R+1,A[R]=x
- popR: R=R-1

Use a BST store pushL,popL by their time, and encode them as -1 and +1. Another one for pushR,popR in the same way.
Then we can know the L, R at the query time t.

Deque:Full

Then we can know the L, R at the query time t.
front: we want to know the array value $A[L]$ at time t. Which is the last pushL which write to it.
Find the last pushL at time $< t$ whose prefix sum is L.
Can be done by simple BST search(store the minimum and maximum prefix sum in a node).

Decomposable Searching Problems

Search Problems:

- insert(x),delete(x) : $S = S \cup \{x\}, S = S \setminus \{x\}$
- query(x,S): only affect by the elements of S.

Search Problems are commutative and have inversion operation.
So they are naturally partial retroactive.
A query is decomposable:$\mathrm{Q}(x,A \cup B) = \mathrm{Q}(x,A) \otimes \mathrm{Q}(x,B)$

Decomposable Searching Problems:Full

How to make decomposable searching problems fully retroactive? Discussion.

Decomposable Searching Problems:Full

We can maintain a timeline, for each element x, we know his existence interval are $[L_x, R_x]$.
Then we use a segment tree, each node maintain a original structure.
Split $[L_x, R_x]$ into $O(\log n)$ nodes in the segment tree, and insert x into those nodes's original structure.
Query at time t can be done by querying every t's ancestor in the segment tree.

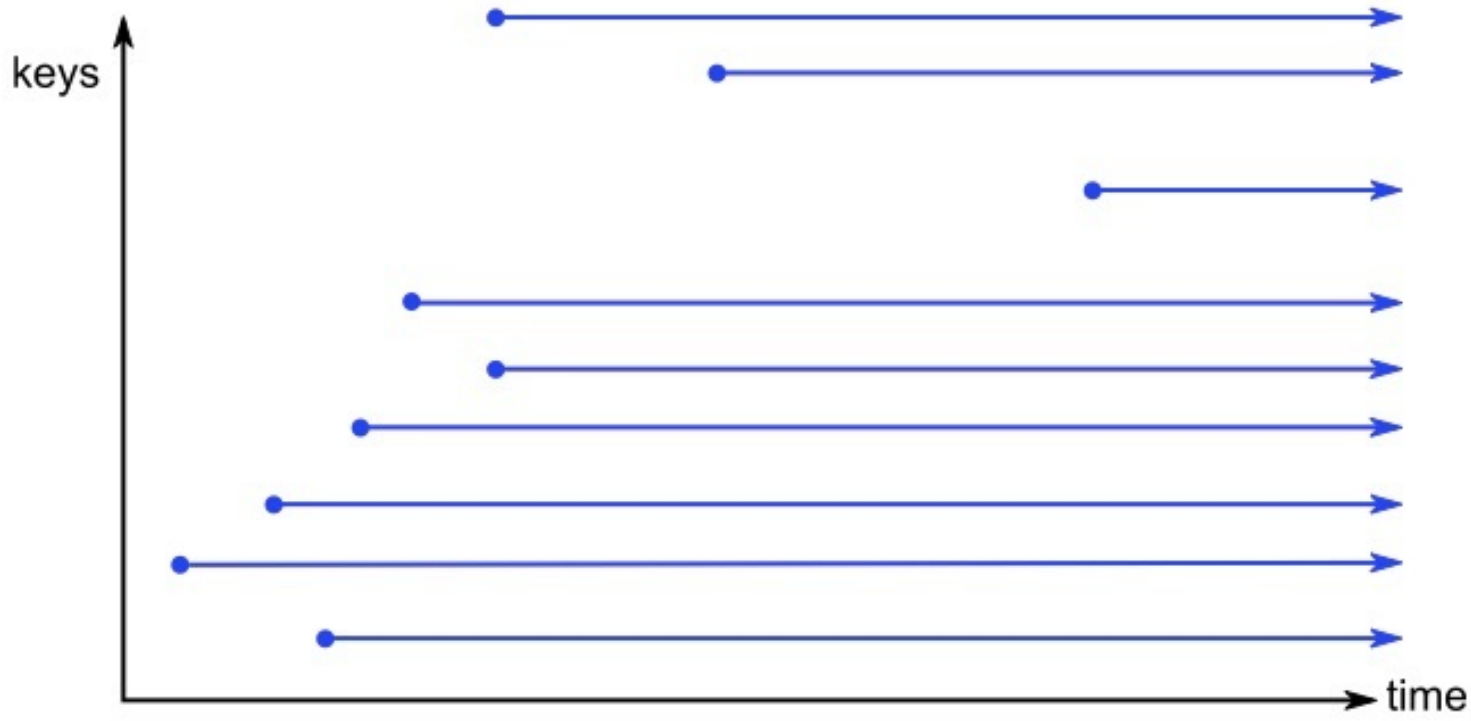
Priority queue

How to make priority queue partial retroactive?
Discussion.

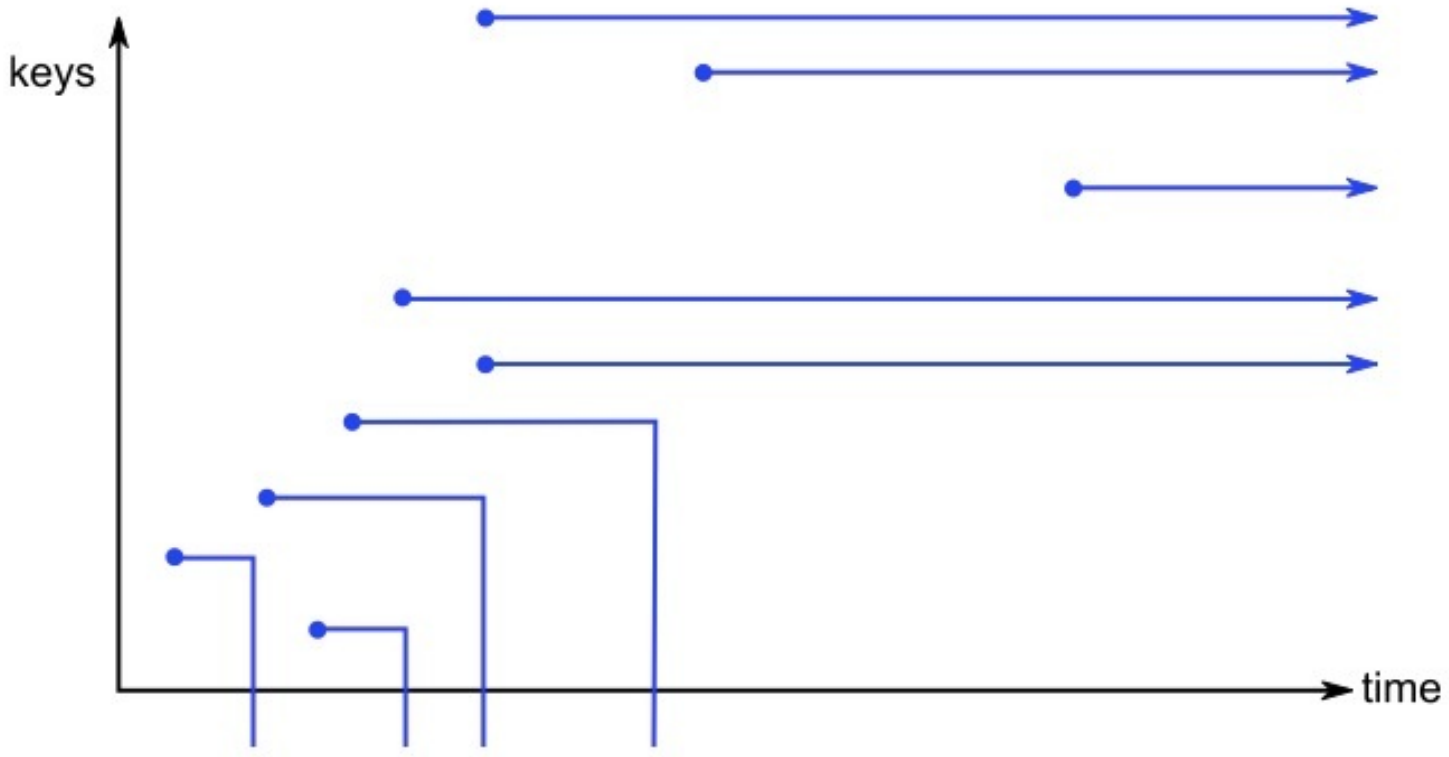
Priority queue

Non-trivial because add an insert in the past can have many effect on succeeding operations.
We can instead using a geometry view.

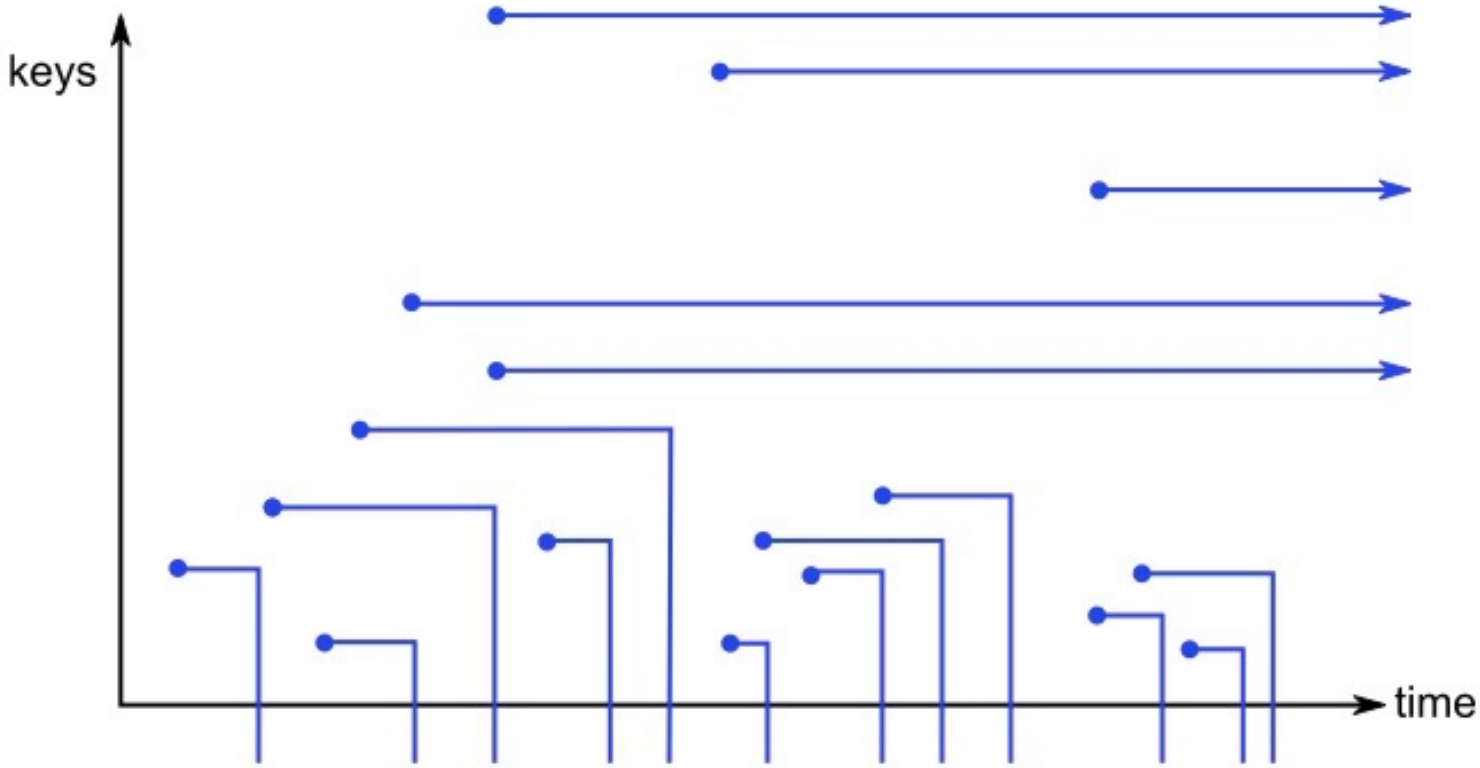
Geometry view: Insertion



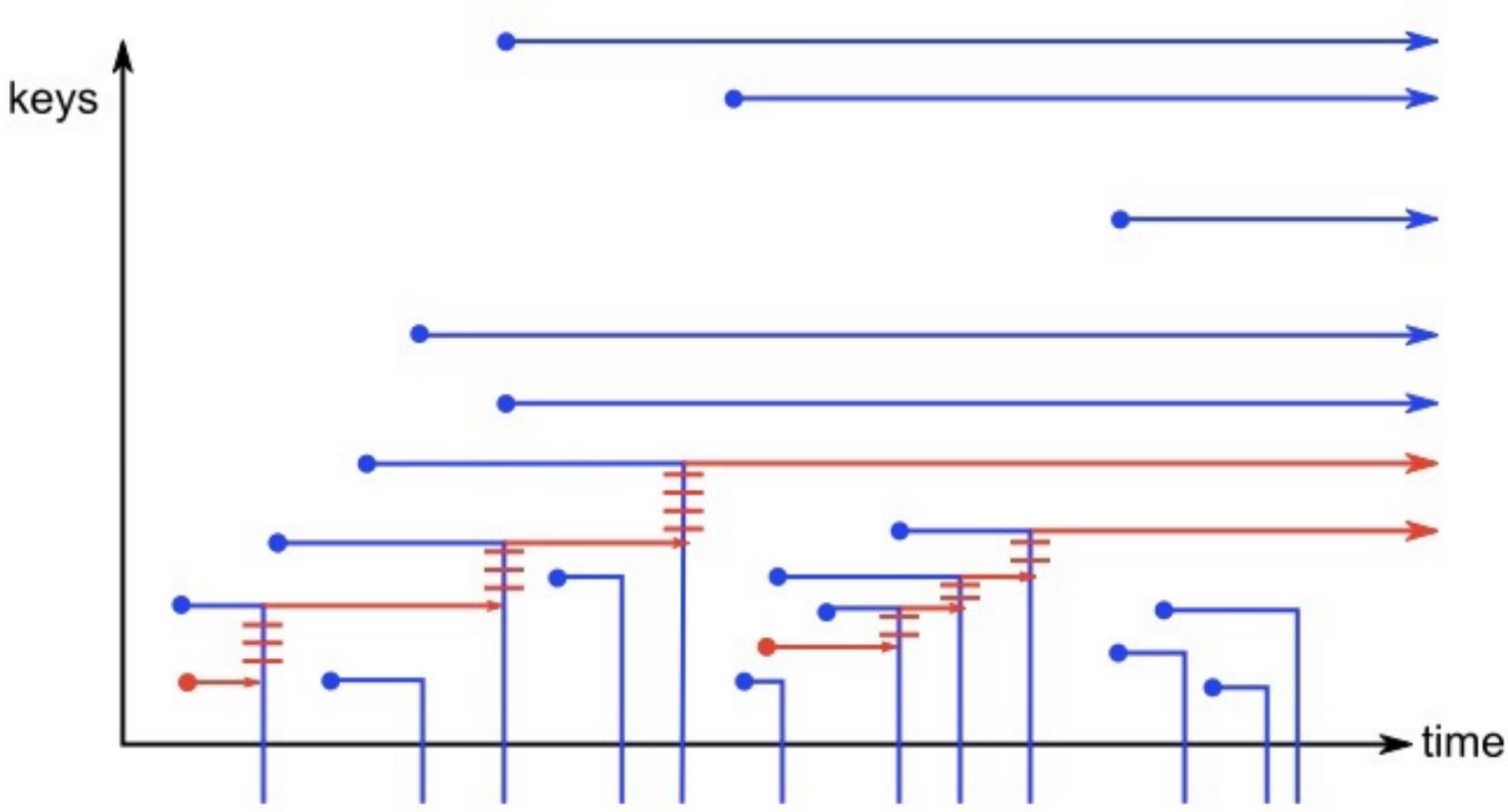
Geometry view: Insertion and Deletion



Geometry view: Insertion and Deletion



Geometry view: Cascading effect of past operation



How to handle this?

When we add an insert in the past, there will be only one elements insert into the current version of the priority queue.
When we add a delete-min in the past, there will be only one elements removed from the current version of the priority queue.
Just find them is ok.

Insert in the past

When we insert element x at the time t, which will happen?

- it just remain, and x is inserted into the current priority queue.
- it got removed sometimes later, so someone got free, and goes on, make some other guys finally entered the current priority queue.

We can see the element which are inserted into the current priority queue is max(x,max$\{x' | x'$ are deleted after time t$\}$).
Taking look at the cascading line(red line), if the largest x' are still deleted, then it lie above the red line, it means that when it got deleted, there is something below it, which is a contradiction(because it is delete-min).

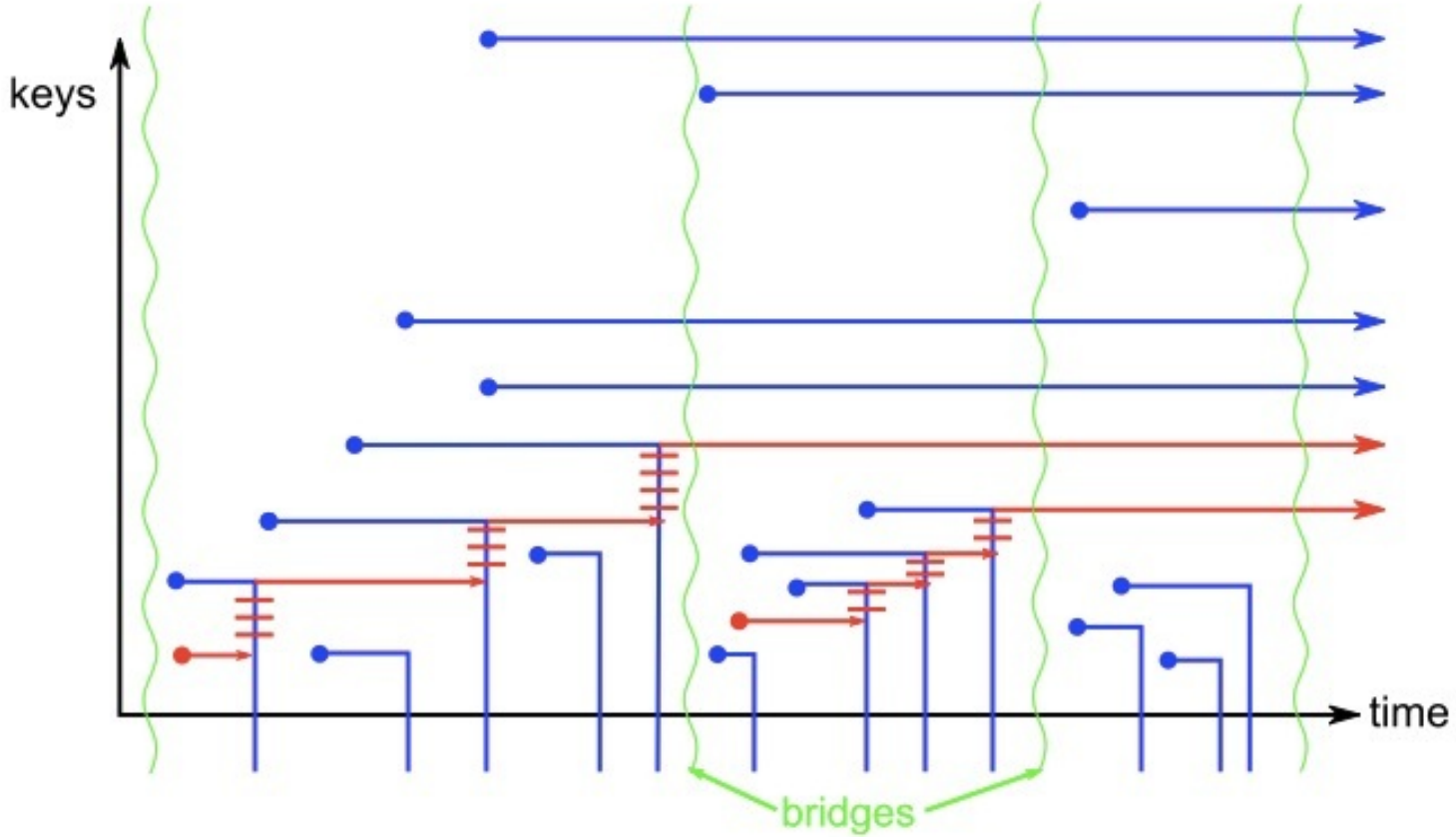
Insert in the past

So we want to know what is max$\{x'|x'$ are deleted after time t$\}$.
It is still hard to approach, because the deleted time will change a lot as an effect of modification.
So can we change deleted time into insertion time? which will not change and the problem get easier.

Bridges

We call time t a bridge if $Q_t \subset Q_{now}$.(Q_i means the priority queue at the time i).
So bridge means that the element remain at time t will continue its way to the end.

Geometry view: Bridges



Insert in the past

Suppose time t' is the preceding bridges of time t.
Then we have:
$\max\{x'|x'$ are deleted after time $t\} = \max\{x'|x' \notin Q_{now}$ and are inserted after time $t'\}$
Let $A = \{x'|x'$ are deleted after time $t\}$,$B = \{x'|x' \notin Q_{now}$ and are inserted after time $t'\}$.
Proof:
$\Rightarrow$ if x' are deleted after time t, then $x' \notin Q_{now}$, so $x' \notin Q_{t'}$, so x' is inserted after time t'. So $A \subset B$, $\max(A) \leq \max(B)$.

Insert in the past

Suppose time t' is the preceding bridges of time t.
Then we have:
$\max\{x'|x'$ are deleted after time $t\} = \max\{x'|x' \notin Q_{now}$ and are inserted after time $t'\}$
Let $A = \{x'|x'$ are deleted after time $t\}, B = \{x'|x' \notin Q_{now}$ and are inserted after time $t'\}$.
Proof:
$\Leftarrow$ if $\max(B) > \max(A)$, let $k = \max(B)$. Then $k \notin A$, so $t' < d_k < t, d_k$ is the deleted time of k. We should know that d_k is not a bridges since there is no bridges between t and t'. So there is an element k', which inserted before d_k and deleted after d_k. $k' \notin Q_{t'}$ because it got deleted, so the inserted time of $k' > t'$, so $k' \in B$ and $k' > k$, which contradict the fact that k is the largest.

delete-min in the past

When we insert a delete-min, who will be finally deleted?
We delete a_1 now, but it is supposed to be deleted at time d_{a_1}, so at that time we will delete a_2 instead of a_1,but it is supposed to be deleted at time d_{a_2}, so at that time we will delete a_3 instead of a_2.
Move on and we finally find a_k such that it will not be deleted.
So we got $a_1 < a_2 < a_3 < \cdots < a_k$, and $a_k \in Q_{now}$.
First we should notice that at time $d_{a_{k-1}}$, everything $< a_k$ is deleted, and a_k remain to Q_{now}, so everything $> a_k$ will too, so the time $d_{a_{k-1}}$ is a bridge.
Also we can notice that from every time between t and $d_{a_{k-1}}$ will not be a bridge because from the chain something got deleted.
So $d_{a_{k-1}}$ is the succeeding bridge of time t. and a_k is the smallest element on that bridge.

delete-min in the past

To summarize, when we insert a delete-min at time t.
We first find the next bridge after t, and delete the smallest element on it.

Other modification

- Delete a delete-min: as same as insert the deleted elements back
- Delete an insert: insert a delete-min right before it got deleted

The Algorithm

What we want is the following:

- Query at the current priority queue(find-min).
- Query $\max\{x' | x' \notin Q_{now}$ and are inserted after time $t'\}$.
- Find the preceding or succeeding bridges of time t.

Query at the current priority queue(find-min)

This is easy, we can use a BST to store the elements of Q_{now}.
Everytime we need to insert something in Q_{now} or delete something from Q_{now}, we simply do it in this BST.
find-min is trivial.

Query $\max\{x' | x' \notin Q_{now}$ and are inserted after time $t'\}$

We again use a BST to store every insert which $\notin Q_{now}$ by time. Then it is simply a suffix maximum value query.

Find the preceding or succeeding bridges of time t.

We use a BST of every operation ordered by time and encode them as following:

- insert $x (x \notin Q_{now})$: $+1$
- delete-min: -1
- insert $x (x \in Q_{now})$: 0

Then we can see that time t is a bridge if every thing before time t in this BST summed to 0.
Then we need to find a t' preceding or succeeding t which has prefix sum 0. It is a trivial BST operation.

The Algorithm: Complexity

We can see that every operation can be done by constant operations on those BSTs.
So the overall complexity is $O(\log n)$. Which is as same as the original complexity in a sense.

Priority Queue: Full

How to make it fully retroactive?
Discussion.

Priority Queue: Full

We will introduce the general methods to make every partial retroactive data structures fully retroactive.
Supposed this data structure has bounded in-degree, then we can make it persistent with no extra time cost.
Suppose there is $O(m)$ operations. We split them into $O(\sqrt{m})$ blocks. Each block has size $O(\sqrt{m})$.
Then we update the operation one by one, and keep the persistent version of it at each block's starting point.
For each query at the previous time t, we first get the version at the t's block's starting time. And do the updates between it and t. Then we get the version at time t.
When we insert operation at time t, we insert it into the corresponding block, and do the retroactive update on those block's starting point after t.
Rebuild the whole data structure when some block's size is larger tphen $2\sqrt{m}$.

Priority Queue: Full

The retroactive priority queue consist of some BST which has bounded in-degree.
So we can make it fully retroactive, and the time complexity is $O(\sqrt{m} \log n)$.

Open Questions

Can we do better in fully retroactive priority queue?