MIT6.851 Partial Persistent Pointer Machine

Jun 07, 2019 » mit6_851 (/category/mit6_851), data_structure (/category/data_structure)

This series (/category/mit6_851) of articles introduce the notes on the lectures MIT6.851 (http://courses.csail.mit.edu/6.851/spring12/) - Advanced Data Structure from Professor Erik Demaine. Here is the first article. All codes are open sourced in my github (https://github.com/ginfung/Advanced_DS). Star 1

Jianfeng Chen

CS.PhD.AI.SE.Infra

(mailto:jchen37@ncsu.edu)

(https://www.facebook.com/jfchen1992)

(https://www.linkedin.com/in/jianfengcs/)

(https://github.com/ginfung)

(/wechat/)

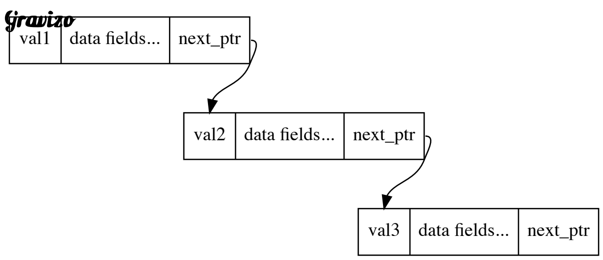
(https://www.slideshare.net/jianfengcs)

1. Pointer Machine

Pointer machine is the model of computation. A pointer machine can be treated as a set of entries. Each entry contains $O(1)$ **fields**. A field can be any type of data, or a pointer to another entry.

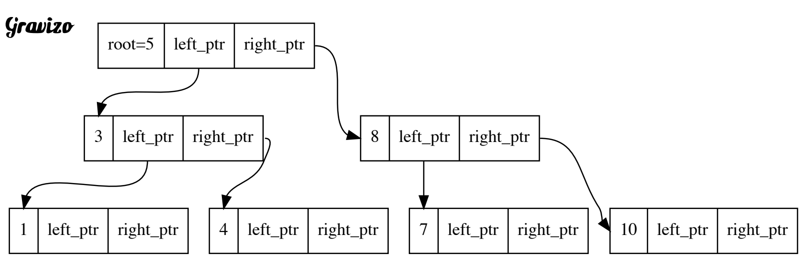
As the first example, the linked list defined like `LinkedList<Integer>` (in java) can be represented via pointer machine where the fields consist of two or more parts

- `val` integer value
- any other `data fields`
- `next_ptr` pointing to next node



Binary search tree (BST) can also be built upon pointer machine. In BST, there are three(3) essential elements in the fields

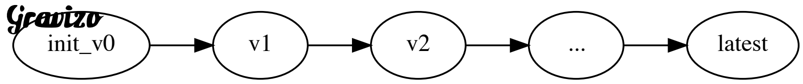
- `val`
- `left_ptr`
- `right_ptr`



2. Partial Persistence

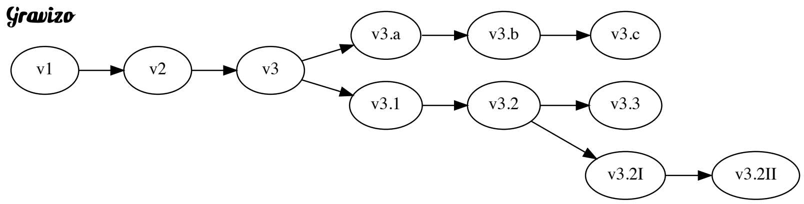
The persistence of some data structure is the ability to keep all versions of that.

Partial persistence: the data structure (DS) can be updated only in latest version. And we can **access (NOT CHANGE)** any field in any version. Versions are linearly ordered as follows,



Example: Assuming we have a partial persistent pointer machines, then we can create a version controlled LinkedList/BST. Please note that all pointers in the LinkedList/BST are part of the fields; like the data fields, all pointers are version controlled.

Full persistence: DS can be updated in any version. This will be discussed in another article. Versions form a tree as



3. An Implementation of Partial Persistance

The implementation is based on Driscoll, Sarnak, Sleator, Tarjan–JCS 1989 (https://www.cs.cmu.edu/~sleator/papers/another-persistence.pdf).

- Any pointer-machine DS with $\leq p = O(1)$ pointers to any node can be made partially persistent
- with $O(1)$ amortized multiplicative overhead and
- $O(1)$ space per change.

3.1 Extended pointer machine

To implement this we need the extend the pointer machine as

- all original data and pointer fields
- p back pointers
- $2p$ mods (modifications), each one is `(version, variable, value)`

Given this, we can declare the pointer machine

```
class PartialPersistentNode():
    def __init__(self, data_vars_tags, ptr_vars_tags, max_pointer_num=1):
        self.p, self.dt, self.pt = max_pointer_num, data_vars_tags, ptr_vars_tags

        self.fields = {var: None for var in self.dt + self.pt}
        self.back = {var: set() for var in self.pt}
        self.mods = list()  # list of tuple (version, field, value)

    def set_back(self, ptr, new, old=None):
            assert ptr in self.pt
            if old is not None and old in self.back[ptr]:
                self.back[ptr].remove(old)
            self.back[ptr].add(new)
```

We let `PartialPersistentNode.dt` and `PartialPersistentNode.pt` as tags of data and pointer fields. For each pointer field, we need a set to hold the nodes pointing to current node.

For convenient, we have a function `set_back` to update the back pointers.

3.2 Read Field in Any Version

Since we put all modifications in `PartialPersistentNode.mods` , To read the `PartialPersistentNode` in any version `v` , we just need to check for the mods with time <= `v` .

```
def read(self, var, v):
    assert var in self.fields.keys()

    res = self.fields[var]
    for (mod_version, mod_field, mod_value) in self.mods:
        if mod_version > v: break
        if mod_field == var: res = mod_value
    return res
```

3.3 Update the Latest Version

To do the update,

- if the `self.mods` is not full, then just append the modification information there
- if mods of `self` is full, we need to
    - create a `new` node with all modifications of fields updated, and all back pointers copied
    - in all nodes `self.fields[ptr] for ptr in self.pt`, their back pointers should be redirected to the `new` node
    - for all nodes pointing to `self`, update their `ptr` field (where `ptr in self.pt`) to `new`. (*NOTE: This may be a recursive process.*)

```
def write(self, now, var, val):
    assert var in self.fields.keys()
    if len(self.mods) <= 2 * self.p:
        self.mods.append((now, var, val))
    else:
        # crate the new node. save the latest value
        new = PartialPersistentNode(self.dt, self.pt, self.p)
        for (_, mod_field, mod_value) in self.mods:
            new.fields[mod_field] = mod_value
        new.fields[var] = val
        new.back = {
            var: set([i for i in self.back[var]])
            for var in self.pt
        }

        # change to backpoints in pointingto
        for ptr in self.pt:
            node_pointing_to = new.fields[ptr]
            node_pointing_to.set_back(ptr, new, self)

        # recursively change pointers to self -> new (found via back pointers)
        for ptr in self.pt:
            for node_point_to_me in new.back[ptr]:
                node_point_to_me.write(now, ptr, new)
```

4. Partial Persistent Linked List

With the `PartialPersistentNode`, we can now implement a partial persistent linked list. The declaration of `LinkedList` can be

```
class LinkedList():
    def __init__(self):
        self.dt, self.pt, self.p = ['value'], ['next_ptr'], 1
        self.root = PartialPersistentNode(self.dt, self.pt, self.p)
        self._timestamp = 0

    def now(self):  # equivalent to  (++now)
        res = self._timestamp
        self._timestamp += 1
        return res

    def set_root_value(self, val):
        self.root.write(self.now(), 'value', val)
```

To simplify, we use the `_timestamp` to track the version.

Reading `LinkedList` is not difficult.

```
def str_(self, v):
        res = ""
        cursor = self.root
        while cursor:
            res += str(cursor.read('value', v)) + '->'
            cursor = cursor.read('next_ptr', v)
        res += 'END'
        return res

    def __str__(self):
        return self.str_(self._timestamp)`
```

To append a node, move the end of the list, then create a `new` node, set up `value` and back pointer – `next_ptr`. Register the `new` node in previous(`cursor` in the following)'s `next_ptr`.

```
def append(self, val):
    at = self.now()
    cursor = self.root
    while True:
        next_node = cursor.read('next_ptr', at)
        if next_node is None:
            break
        else:
            cursor = next_node

    new = PartialPersistentNode(self.dt, self.pt, self.p)
    new.write(at, 'value', val)
    new.set_back('next_ptr', cursor)
    cursor.write(at, 'next_ptr', new)
    return self
```

Similar operations for inserting,

```
def insert_after_ith_node(self, i, val):
        at = self.now()
        cursor = self.root
        for _ in range(i):
            cursor = cursor.read('next_ptr', at)
            if cursor is None:
                self.append(val)
                return self
        next_ = cursor.read('next_ptr', at)

        new = PartialPersistentNode(self.dt, self.pt, self.p)
        new.write(at, 'value', val)

        cursor.write(at, 'next_ptr', new)
        new.write(at, 'next_ptr', next_)

        next_.set_back('next_ptr', new, cursor)
        new.set_back('next_ptr', cursor)

        return self
```

Updating the value of some node can be

```
def modify_ith_node_val(self, i, val):
    assert i >= 0
    at = self.now()
    cursor = self.root
    for _ in range(i):
        cursor = cursor.read('next_ptr', at)
        if cursor is None:
            self.append(val)
            return self
    cursor.write(at, 'value', val)

    return self
```

To delete some node, carefully update the back pointers and `next_ptr` shoud be fine :)

```
    def delete_ith_node(self, i):
        assert i > 0, "Assuming do not delete the root"
        at = self.now()
        cursor = self.root
        for _ in range(i - 1):  # move the one before what
            cursor = cursor.read('next_ptr', at)
            if cursor is None: return self

        next_ = cursor.read('next_ptr', at)
        next_next = next_.read('next_ptr', at)
        cursor.write(at, 'next_ptr', next_next)
        next_next.set_back('next_ptr', cursor, next_)

        return self
```

The following shows the demo

```
if __name__ == '__main__':
    L = LinkedList()
    L.set_root_value(1)
    L.append(2).append(3).append(4).append(5)
    cp1 = L.now()  # checkpoint 1
    print(f'Current time = {cp1}, Latest linked list shows as {L}')

    L.modify_ith_node_val(2, 20)
    cp2 = L.now()
    print(f'Current time = {cp2}, Latest linked list shows as {L}')

    L.insert_after_ith_node(2, 22)
    cp3 = L.now()
    print(f'Current time = {cp3}, Latest linked list shows as {L}')

    L.modify_ith_node_val(1, 10)
    cp4 = L.now()
    print(f'Current time = {cp4}, Latest linked list shows as {L}')

    L.delete_ith_node(2)
    cp5 = L.now()
    print(f'Current time = {cp5}, Latest linked list shows as {L}')

    print("\n\nRolling back to earliest check points")
    for cp in [cp5, cp4, cp3, cp2, cp1]:
        print(f'In {cp}, list shows as {L.str_(cp)}')
```

Output as

```
>>> STDOUT
Current time = 5, Latest linked list shows as 1->2->3->4->5->END
Current time = 7, Latest linked list shows as 1->2->20->4->5->END
Current time = 9, Latest linked list shows as 1->2->20->22->4->5->END
Current time = 11, Latest linked list shows as 1->10->20->22->4->5->END
Current time = 13, Latest linked list shows as 1->10->22->4->5->END

Rolling back to earliest check points
In 13, list shows as 1->10->22->4->5->END
In 11, list shows as 1->10->20->22->4->5->END
In 9, list shows as 1->2->20->22->4->5->END
In 7, list shows as 1->2->20->4->5->END
In 5, list shows as 1->2->3->4->5->END
```

A complete code in this article can be found at here (https://github.com/ginfung/Advanced_DS/blob/master/time_travel/partial.py).

« Hello World (/front-end/2017/10/07/hello-world.html)